Wesleyan ♦ University

COMPUTER-CHECKED RECURRENCE EXTRACTION FOR FUNCTIONAL PROGRAMS

By

Bowornmet Hudson

Faculty Advisor: Daniel Licata

A Dissertation submitted to the Faculty of Wesleyan University in partial fulfillment of the requirements for the degree of Master of Arts

Middletown, Connecticut **April, 2016**

I suddenly remember James Joyce and stare at the waves realizing "All summer you were sitting here writing the so called sound of the waves not realizing how deadly serious our life and doom is, you fool, you happy kid with a pencil, dont you realize you've been using words as a happy game— all those marvelous skeptical things you wrote about graves and sea death it's ALL TRUE YOU FOOL! Joyce is dead! The sea took him! it will take YOU!"

– Big Sur, *Jack Kerouac*

Acknowledgements

To my advisor, Dan Licata, for having me work with him these past two years, and for giving me an interesting and fruitful project to work on. The acknowledgements for Dan from my previous thesis should be copied and pasted here, but I'll add a little something extra to this one. I remember in our first email exchange (before taking his Agda course), he asked me if I'd taken a PL course and I thought PL was a language and not an abbreviation for 'Programming Languages.' Boy, was that embarrassing, and boy, was I wrong. Well, here we are now, two theses later – I think I'm getting a little better at playing the game.

To my committee, Norman Danner and Olivier Hermant, for taking the time to evaluate this work, and for being two computer scientists I look up to. Norman and Olivier have contributed immensely to my academic growth, and have given me many new ways to think about logic and computer science. Both are also insanely fun to hang out with. Special thanks to Norman, for leading the project we are contributing to, and for being the reason I decided to major in computer science as an undergrad.

To my colleagues, Justin and Ted, for grinding through this year with me and for always pushing me to find a better solution. It's been a wild ride. By the way, what happens in ESC345 stays in ESC345.

To my friends, Keonmin, Rohit, and Will, for everything we've done, and for always being there to pick me up when I needed the motivation to carry on. I couldn't have asked for a better group of friends to help me get through this year.

To my parents, for everything. Thank you for being patient with me and for believing in me. You were right.

This thesis is dedicated to my parents.

Abstract

This thesis continues work towards a computer-checked system for automatic complexity analysis of functional programs. Following [DLR15], we extract cost information from a source language to a monadic metalanguage which we call the complexity language, and give a denotational semantics to the complexity language which interprets types as preorders and terms as monotone maps between preorder structures. Building upon the work presented in [Hud15], we extend the source language with more types, and prove that the complexity language is sound relative to its interpretation. In addition, we observe that the recurrences we obtain using our system resemble recurrences obtained manually. We implement all of our work in Agda, a dependently-typed programming language and proof assistant.

Contents

CHAPTER 1

Introduction

Time complexity analysis for a recursive program can generally be broken down into two steps: extracting a recurrence relation that describes the program's running time in terms of the size of its input, then reducing the recurrence to obtain a closed form and asymptotic bound on the running time. This can be an arduous process as it is usually computed by hand; this is especially true for large, elaborate pieces of code. [Hof11] remarks that "derivation of precise bounds by hand appears to be unfeasible in practice in all but the simplest cases."

This thesis describes the use of Agda [Nor07], a dependently-typed programming language and proof assistant, to develop a system to automate the process of recurrence extraction for functional programs. In particular, we show that

> *It is possible to extract and formally reason about time complexity properties of functional programs using proof assistants.*

The work presented in this thesis stems from two particular areas of interest. The first, more obvious area is automated complexity analysis, which aims to develop methods to automatically compute the running time of a program. The second is the use of proof assistants to tackle problems in programming language theory, which includes automated complexity analysis.

1. Automated Complexity Analysis

Automated complexity analysis has been an active area of research for several decades, and a large collection of literature investigating this topic has already been produced. Following Wegbreit's Metric system, first described in his seminal contribution [Weg75], much of the work, including ours, seeks to extract and solve recurrences from programs, which in turn describe their running time in terms of input size.

The main focus of this work is the automatic extraction of recurrences from functional programs. Our approach to recurrence extraction is inspired by [DLR15], which is in turn inspired by [DPR13]: we extract cost information from source programs by applying a transformation to a monadic metalanguage [Mog91], which we call the complexity language. [DLR15] also defines a model of the complexity language which interprets types as preorders and terms as monotone maps between elements of preorders. In this thesis, we define a similar denotational semantics, and prove that the complexity language is sound relative to it.

Following [Hof11], our method must appeal to four criteria:

(1) *Range*: the range of programs we can analyze with our system
(2) *Precision*: the accuracy of the bounds we produce
(3) *Efficiency*: how long it takes to compute bounds
(4) *Verifiability*: how easy it is to verify the bounds are correct

Range. The range of programs our system can analyze is not only limited by the syntax of the source language we decide to implement, but also by the framework in which we implement it. At the moment, we do not consider coinductive types and non-terminating computations, but in future work, we could make use of Agda's delay monad to do so.

Precision. In [Hud15], we prove that the costs extracted from source programs using our method are an upper bound on the costs specified by the source language operational semantics. This is a basic correctness criterion – we also observe that the recurrences obtained by running algorithms through the denotational semantics resemble recurrences obtained by manual means. The soundness of the complexity language with respect to its denotational semantics also attests to the accuracy of our system.

Efficiency. Aside from hardware, the efficiency of our system relies on the implementation of the translation function from source to complexity, which is the mechanism for extracting cost information from source programs. A direct implementation of the translation function given by [DLR15] results in an exponential inflation in the size of translated terms, because it appeals to recursive calls repeatedly. This drastically reduces the speed at which we extract recurrences; we discuss optimizations to the direct implementation to avoid this problem.

Verifiability. Implementing all of the languages and proofs about our system in a proof assistant guarantees correctness of the language specification and bounds produced.

2. Agda

We implement all of our work in Agda [Nor07], a dependently-typed functional programming language and *proof assistant*. Agda's nature as a proof assistant allows users to formalize mathematical structures and prove theorems about them. In this section, we briefly review some Agda syntax which may help the reader understand the code given in this thesis, referring the reader to the Agda Wiki (`http://wiki.portal.chalmers.se/agda/`). We also refer the reader to [Geu09] for a detailed account of the history and fundamental ideas behind proof assistants.

2.1. Inductive Types. Inductive types are defined in Agda as datatypes:

```
data Nat : Set where
  Z : Nat
  S : Nat → Nat
```

2.2. Infix Operators. Infix operators are defined using underscores:

```
_⩽_ : Nat → Nat → Bool
Z ⩽ n = True
S m ⩽ Z = False
S m ⩽ S n = m ⩽ n
```

2.3. Equation Chains. For any terms a and c, the basic Agda syntax for an equation chain to show $a = c$ is

```
a =⟨ p1 ⟩ b =⟨ p2 ⟩ c ■
```

where b is some intermediary term, and p1 and p2 are proofs that $a = b$ and $b = c$ respectively.

Proof assistants are powerful tools which can be used to investigate a wide range of topics, and have already been used to formalize a wide range of topics in mathematics, such as graph

theory, ([Gon07]), algebra ([GAA+13]), and homotopy type theory ([Uni13]). Using proof assistants to approach problems has the potential for many new insights into mathematics, including proof theory, programming language theory, and category theory.

3. Outline and Contributions

The overall aim of this project is to develop a formal system for automated complexity analysis in Agda. The main mechanism for extracting cost information from source programs is a monadic translation $||\cdot||$ into a complexity language, where we can reason about program costs directly without needing to appeal to a denotational semantics. In Chapter 3, we prove that the costs extracted are an upper bound on the costs specified by the source language operational semantics.

Since the translation yields exact recurrences and doesn't do any abstraction of sizes or values, it would be useful to be able to compute more relaxed upper bounds on program costs and manipulate the recurrences into closed forms, from which we can deduce an asymptotic bound on the running time of the original source program. To enable this, we equip the complexity language with an abstract preorder judgement $\_ \leqslant s \_$ which takes the place of a traditional operational semantics and specifies ordering on terms. With our end goal in mind, we must ensure that all the recurrences we extract are monotone with respect to the size of their input. As a concrete example, consider insertion sort, which we can define in Agda as

```
isort : (l : List Nat) → List Nat
isort [] = []
isort (x :: xs) = insert x (isort xs)
  where
    insert : (el : Nat) (l : List Nat) → List Nat
    insert el [] = el :: []
    insert el (x :: xs) with el ⩽ x
    ... | True = el :: x :: xs
    ... | False = x :: insert el xs
```

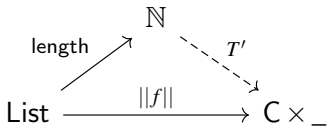
and where $\leqslant$ on $\mathbb{N}$ is defined in the usual way. The translation of a source language implementation of isort into the complexity language gives us an exact recurrence of the form

$$T_{\mathsf{isort}}([]) = c_0$$
$$T_{\mathsf{isort}}(\mathsf{x} :: \mathsf{xs}) = c_1 + T_{\mathsf{insert}}(\mathsf{xs}) + T_{\mathsf{isort}}(\mathsf{xs})$$

for fixed integer constants c_0 and c_1. We can also write a recurrence in the complexity language

$$T'_{\mathsf{isort}}(0) = c_0$$
$$T'_{\mathsf{isort}}(n) = c_1 + T'_{\mathsf{insert}}(n-1) + T'_{\mathsf{isort}}(n-1)$$

which abstracts the size of the argument to isort. If we can show $T_{\mathsf{isort}}(\mathsf{xs}) \leqslant \mathsf{s}\ T'_{\mathsf{isort}}(\mathsf{length}\ \mathsf{xs})$, where length is a function which computes the length of a list, then we can conclude that the extracted recurrence is indeed monotone with respect to the size of its argument. For a function f which acts on lists, this process of ensuring monotonicity of recurrences in the complexity language can be visualized in terms of the following commutative diagram:



where T' is a recurrence which abstracts the size of the argument to f, and the composite $\mathsf{length} \circ T'$ is an upper bound on the exact recurrence obtained by $||\ f\ ||$.

This desire for flexibility in reasoning about costs motivates our choice of denotational semantics for the complexity language, where we interpret complexity types and expressions as preorders and monotone functions between preorders. In addition, the asymptotic bounds which we would like to compute from recurrences are all monotonic functions, so any interpretation of a complexity recurrence or closed form should be monotonic as well. In Chapter 4, we prove that the complexity language is sound with respect to the given interpretation.

The work presented in this thesis builds upon the work of [Hud15]. Notably, we extend the source language, and prove that the complexity language is sound relative to the interpretation we give. Our development consists of more than 5,000 lines of Agda. All of the source

code for the work in this thesis can be found at `https://github.com/benhuds/Agda/tree/master/complexity/complexity-final/submit`. Chapters 2 and 3 introduce the source and complexity languages, along with the translation from the former to the latter. Chapter 4 gives a denotational semantics to the complexity language which interprets types as preorders and terms as monotone maps between preorders, and Chapter 5 presents the results of running several algorithms through our system. Chapter 6 discusses related work in our area of study, focusing on past approaches to recurrence extraction, and the use of proof assistants in complexity analysis. Chapter 7 concludes the thesis with a summary of our contributions, as well as ideas for future work.

CHAPTER 2

Source and Complexity Languages

In this chapter, we introduce the source language for our complexity analysis system, and the language where the abstract cost measures specified by the source language operational semantics are made concrete, which we call the complexity language. The costs specified by the source language operational semantics are made explicit by a translation function $||\cdot||$, which we will discuss in the next chapter.

1. Source Language

1.1. Syntax. The source language is a simply-typed λ-calculus, augmented with product types, natural numbers, lists, and booleans. In addition, we use suspensions to avoid unnecessary computations when unfolding datatype recursors. In our Agda implementation, we represent contexts as lists of types, and variables in contexts as de Bruijn indices[dB72]. The source language types, contexts, and variables are shown in Figure 1; the typing rules are shown in Figure 2. We omit the rules for the closed values of the source language, which are denoted by a datatype val.

1.2. Semantics. We define an operational cost semantics to describe the evaluation of source expressions to values. First introduced in [BG96], cost semantics provide a notion of time complexity to source expressions by adding a cost measure to each evaluation relation. We denote this cost measure with a datatype Cost:

```
data Cost : Set where
  0 c : Cost
  1 c : Cost
  _+c_ : Cost → Cost → Cost
```

```
data Tp : Set where
  unit : Tp
  nat : Tp
  susp : Tp → Tp
  _->s_ : Tp → Tp → Tp
  _×s_ : Tp → Tp → Tp
  list : Tp → Tp
  bool : Tp

Ctx = List Tp

data _∈_ : Tp → Ctx → Set where
  i0 : ∀ {Γ τ} → τ ∈ τ :: Γ
  iS : ∀ {Γ τ τ1} → τ ∈ Γ → τ ∈ τ1 :: Γ
```

FIGURE 1. Source language types, contexts, and variables

In the semantics, we write $e \downarrow^n v$, meaning "e evaluates to a value v in n steps," where n is of type Cost. Part of the operational semantics is given in Figure 3.

In Agda, we define substitution as a function which maps variables in a context Γ' to expressions derived from another (not necessarily different) context Γ:

```
sctx : Ctx → Ctx → Set
sctx Γ Γ' = ∀ {τ} → τ ∈ Γ' → Γ |- τ
```

This intrinsic style, described in [BHKM12], is attractive in that it does not require induction on the length of contexts and is thus more concise. However, the drawback in defining substitution as a single function is that function extensionality is not built into Agda. Tuple extensionality, on the other hand, comes for free in Agda, so verifying properties about substitution defined in the extrinsic brute-force manner can be more straightforward.

Define the application of a substitution θ to a source term as follows:

```
data _|-_ : Ctx → Tp → Set where
  unit : ∀ {Γ} → Γ |- unit
  var : ∀ {Γ τ} → τ ∈ Γ → Γ |- τ
  z : ∀ {Γ} → Γ |- nat
  suc : ∀ {Γ} → Γ |- nat → Γ |- nat
  rec : ∀ {Γ τ} → Γ |- nat → Γ |- τ → (nat :: (susp τ :: Γ)) |- τ → Γ |- τ
  lam : ∀ {Γ τ ρ} → (ρ :: Γ) |- τ → Γ |- (ρ ->s τ)
  app : ∀ {Γ τ1 τ2} → Γ |- (τ2 ->s τ1) → Γ |- τ2 → Γ |- τ1
  prod : ∀ {Γ τ1 τ2} → Γ |- τ1 → Γ |- τ2 → Γ |- (τ1 ×s τ2)
  delay : ∀ {Γ τ} → Γ |- τ → Γ |- susp τ
  force : ∀ {Γ τ} → Γ |- susp τ → Γ |- τ
  split : ∀ {Γ τ τ1 τ2} → Γ |- (τ1 ×s τ2) → (τ1 :: (τ2 :: Γ)) |- τ → Γ |- τ
  nil : ∀ {Γ τ} → Γ |- list τ
  _::s_ : ∀ {Γ τ} → Γ |- τ → Γ |- list τ → Γ |- list τ
  listrec : ∀ {Γ τ τ'} → Γ |- list τ → Γ |- τ' → (τ :: (list τ :: (susp τ' :: Γ))) |- τ' → Γ |- τ'
  true : ∀ {Γ} → Γ |- bool
  false : ∀ {Γ} → Γ |- bool
```

FIGURE 2. Source language typing rules

```
subst : ∀ {Γ Γ' τ} → Γ' |- τ → sctx Γ Γ' → Γ |- τ
subst unit Θ = unit
subst (var x) Θ = Θ x
subst z Θ = z
subst (suc e) Θ = suc (subst e Θ)
subst (rec e e1 e2) Θ = rec (subst e Θ) (subst e1 Θ) (subst e2 (s-extend (s-extend Θ)))
subst (lam e) Θ = lam (subst e (s-extend Θ))
```

```
mutual
  data evals : {τ : Tp} → [] |- τ → [] |- τ → Cost → Set where
    pair-evals : ∀ {n1 n2}
      → {τ1 τ2 : Tp} {e1 v1 : [] |- τ1} {e2 v2 : [] |- τ2}
      → evals e1 v1 n1
      → evals e2 v2 n2
      → evals (prod e1 e2) (prod v1 v2) (n1 +c n2)
    lam-evals : ∀ {ρ τ} {e : (ρ :: []) |- τ}
      → evals (lam e) (lam e) 0 c
    rec-evals : ∀ {n1 n2}
      → {τ : Tp} {e v : [] |- nat} {e0 v' : [] |- τ} {e1 : (nat :: (susp τ :: [])) |- τ}
      → evals e v n1
      → evals-rec-branch e0 e1 v v' n2
      → evals (rec e e0 e1) v' (n1 +c (1 c +c n2))
    delay-evals : {τ : Tp} {e : [] |- τ}
      → evals (delay e) (delay e) 0 c
    force-evals : ∀ {n1 n2}
      → {τ : Tp} {e' v : [] |- τ} {e : [] |- susp τ}
      → evals e (delay e') n1
      → evals e' v n2
      → evals (force e) v (n1 +c n2)
  data evals-rec-branch {τ : Tp} (e0 : [] |- τ)
    (e1 : (nat :: (susp τ :: [])) |- τ) : (e : [] |- nat) (v : [] |- τ) → Cost → Set where
      evals-rec-z : ∀ {v n} → evals e0 v n → evals-rec-branch e0 e1 z v n
      evals-rec-s : ∀ {v v' n} → evals (subst e1 (lem4 v (delay (rec v e0 e1)))) v' n
        → evals-rec-branch e0 e1 (suc v) v' n
```

FIGURE 3. Source operational cost semantics (partial implementation)

```
subst (app e e₁) Θ = app (subst e Θ) (subst e₁ Θ)
subst (prod e1 e2) Θ = prod (subst e1 Θ) (subst e2 Θ)
subst (delay e) Θ = delay (subst e Θ)
subst (force e) Θ = force (subst e Θ)
subst (split e e₁) Θ = split (subst e Θ) (subst e₁ (s-extend (s-extend Θ)))
subst nil Θ = nil
subst (x ::s xs) Θ = subst x Θ ::s subst xs Θ
subst true Θ = true
subst false Θ = false
subst (listrec e e₁ e₂) Θ =
  listrec (subst e Θ) (subst e₁ Θ) (subst e₂ (s-extend (s-extend (s-extend Θ))))
```

In our implementation, we generalize to simultaneous substitutions, and write $\theta = [t_1/x_1, ..., t_n/x_n]$ to mean a substitution of all variables $x_1, ..., x_n$ in a context Γ for terms $t_1, ..., t_n$. We write $\theta : \Gamma' \to \Gamma$ for a substitution θ to specify its domain and codomain. The interesting cases throughout our work are the ones where we have to apply a substitution under a binder (lam, rec, listrec). For these cases, we need a lemma which allows us to apply substitutions under parallel extension of contexts:

```
s-extend : ∀ {Γ Γ' τ} → sctx Γ Γ' → sctx (τ :: Γ) (τ :: Γ')
s-extend Θ i0 = var i0
s-extend Θ (iS x) = wkn (Θ x)
```

s-extend is derivable using another auxiliary function, wkn (the structural property of weakening), and shifts all context variables in a term. We define weakening as a substitution of variables for variables – this is simply the notion of renaming, which we also implement intrinsically:

```
rctx : Ctx → Ctx → Set
rctx Γ Γ' = ∀ {τ} → τ ∈ Γ' → τ ∈ Γ
```

As with substitutions, we write $\rho : \Gamma' \to \Gamma$ for a renaming ρ to specify its domain and codomain. Applying a renaming ρ to a term is defined in a similar manner to subst. We can define operations which compose substitutions and renamings:

```
_rr_ : ∀ {A B C} → rctx A B → rctx B C → rctx A C
_rr_ ρ1 ρ2 x = (ρ1 o ρ2) x

_rs_ : ∀ {A B C} → rctx A B → sctx B C → sctx A C
_rs_ ρ Θ x = ren (subst (var x) Θ) ρ

_ss_ : ∀ {A B C} → sctx A B → sctx B C → sctx A C
_ss_ Θ1 Θ2 x = subst (subst (var x) Θ2) Θ1

_sr_ : ∀ {A B C} → sctx A B → rctx B C → sctx A C
_sr_ Θ ρ x = subst (ren (var x) ρ) Θ
```

However, given the intrinsic encoding of renamings and substitutions, we need to prove commutative properties for each of the above operations when applied to terms:

```
rr-comp : ∀ {Γ Γ' Γ'' τ} → (ρ1 : rctx Γ Γ') → (ρ2 : rctx Γ' Γ'') → (e : Γ'' |- τ)
  → (ren (ren e ρ2) ρ1) ≡ (ren e (ρ1 rr ρ2))

rs-comp : ∀ {A B C τ} → (ρ : rctx C A) (Θ : sctx A B) (e : B |- τ)
  → ren (subst e Θ) ρ ≡ subst e (ρ rs Θ)

sr-comp : ∀ {Γ Γ' Γ'' τ} → (Θ : sctx Γ Γ') → (ρ : rctx Γ' Γ'') → (e : Γ'' |- τ)
  → (subst (ren e ρ) Θ) ≡ subst e (Θ sr ρ)

ss-comp : ∀ {A B C τ} → (Θ1 : sctx A B) (Θ2 : sctx B C) (e : C |- τ)
  → subst e (Θ1 ss Θ2) ≡ subst (subst e Θ2) Θ1
```

All of the above lemmas are proved by induction on the structure of the term e. We also have lemmas which conclude that a substitution can be weakened by one or more terms in the domain; we omit their proofs for brevity:

```
lem3' : ∀ {Γ Γ' τ} → sctx Γ Γ' → Γ |- τ → sctx Γ (τ :: Γ')
lem4' : ∀ {Γ Γ' τ1 τ2} → sctx Γ Γ' → Γ |- τ1 → Γ |- τ2 → sctx Γ (τ1 :: (τ2 :: Γ'))
lem5' : ∀ {Γ Γ' τ1 τ2 τ3} → sctx Γ Γ' → Γ |- τ1 → Γ |- τ2 → Γ |- τ3 → sctx Γ (τ1 :: (τ2 :: (τ3 :: Γ')))
```

The difficulty in using our intrinsic approach became most apparent when proving lemmas which use properties of composition of renaming and substitution on terms, as we needed to prove function extensionality for each lemma. Take for example the following:

LEMMA 2.1 (Substitution composition). *If $x \notin Dom(\theta)$, then $e[\theta, x/x][e_1/x] = e[\theta, e_1/x]$*

PROOF. The proof is stated and proved in Agda as follows:

```
subst-compose : ∀ {Γ Γ' τ τ1} (Θ : sctx Γ Γ') (v : Γ |- τ) (e : (τ :: Γ' |- τ1))
  → subst (subst e (s-extend Θ)) (q v) ≡ subst e (lem3' Θ v)
subst-compose Θ v e =
  subst (subst e (s-extend Θ)) (q v) =⟨ ! (subst-ss (q v) (s-extend Θ) e) ⟩
  subst e (q v ss s-extend Θ) =⟨ ap (subst e) (subst-compose-lemma v Θ) ⟩
  (subst e (lem3' Θ v) ■)
```

□

Although the statement and proof of the above lemma can be expressed in a few lines of Agda, the main sub-lemma subst-compose-lemma itself requires two more sub-lemmas which prove its extensionality:

```
fuse1 : ∀ {Γ Γ' τ τ'} (v : Γ |- τ') (Θ : sctx Γ Γ') (x : τ ∈ Γ') → (q v ss q• Θ) x ≡ Θ x
fuse1 v Θ x = subst (ren (Θ x) iS) (q v) =⟨ sr-comp (q v) iS (Θ x) ⟩
  subst (Θ x) ids =⟨ ! (subst-id (Θ x)) ⟩
  (Θ x ■)

subst-compose-lemma-lemma : ∀ {Γ Γ' τ τ'} (v : Γ |- τ') (Θ : sctx Γ Γ') (x : τ ∈ τ' :: Γ')
  → _==_ {_} {Γ |- τ} (_ss_ (q v) (s-extend Θ) x) (lem3' Θ v x)
subst-compose-lemma-lemma v Θ i0 = Refl
```

```
subst-compose-lemma-lemma v Θ (iS x) =
  subst (wkn (subst (var x) Θ)) (lem3' ids v) =⟨ subst-ss (q v) (q• Θ) (var x) ⟩
  subst (var x) (q v ss q• Θ) =⟨ fuse1 v Θ x ⟩
  Θ x ■

subst-compose-lemma : ∀ {Γ Γ' τ} (v : Γ |- τ) (Θ : sctx Γ Γ')
  → _==_ {_} {sctx Γ (τ :: Γ')} ((q v) ss (s-extend Θ)) (lem3' Θ v)
subst-compose-lemma v Θ = λ=i (λ τ → λ= (λ x → subst-compose-lemma-lemma v Θ x))
```

More than half of the code required to define the source language consists of these sub-lemmas.

2. Complexity Language

The complexity language is designed to be a metalanguage where we can reason about program costs directly without having to appeal to a denotational semantics. To this end, we feature an abstract preorder judgement $\_ \leqslant \mathsf{s}\_$ which specifies ordering on terms, in place of a traditional operational semantics. This will allow us to 'massage' recurrences into closed forms and asymptotic bounds on the running time of the original source programs. We therefore do not restrict ourselves to exact costs: the type system of the complexity language permits increased flexibility when reasoning about program costs and complexity language terms.

2.1. Syntax. The complexity language types are almost identical to the source language types, with the exception of three distinguishing additions:

(1) We have a base type C, which we use to interpret source costs specified by the source language operational semantics.

(2) We have a second type with which to interpret natural numbers, rnat, which can be interpreted as a 'monotone' version of the natural numbers – its recursor is equipped with a proof that the result of its base case is 'less than' the result of its successor branch. rnat will come in handy when we need to prove that the recurrences we extract with $||\cdot||$ are monotone with respect to the size of their inputs.

(3) We have 'max' types CTpM which add the notion of maximums to some complexity language base types. This permits weakening of assumptions about program costs and therefore allow us to find more general time bounds for functions. To accompany max types, we have a complexity term

$$\mathsf{max} : \forall\,\{\Gamma\ \tau\} \to \mathsf{CTpM}\ \tau \to \Gamma \mid\!\text{-}\ \tau \to \Gamma \mid\!\text{-}\ \tau \to \Gamma \mid\!\text{-}\ \tau$$

which allows us to perform operations like find the lowest upper bound of two recurrences within the syntax of the complexity language. In contrast to our intended interpretation of complexity types as preorders, the interpretation of max types will be preorders with maximums: that is, preorders equipped with an operation to compute the maximum of two values within the preorder, and proofs that the operation preserves ordering and least upper bounds.

In addition to the complexity language types, we introduce let bindings as a primitive term of the complexity language for the main purpose of optimizing the translation function $||\cdot||$. We discuss this in more detail in the next chapter. We write $\times$c, ->c, and ::c to distinguish complexity from source syntax. The complexity language types, terms, and max types are shown in Figure 4.

2.2. Semantics. The intention of the complexity language as a vehicle for reasoning about costs is the driving force behind the complexity language semantics: we have an abstract preorder judgement _$\leqslant$s_ which specifies ordering on terms, and plays an analogous role to a traditional operational semantics. We aren't concerned with the evaluation steps of complexity terms because we already obtain all the information we desire from the source to complexity translation $||\cdot||$. Instead, we hope to use the preorder judgement to solve recurrences, by manipulating them into a form which is equivalent to a closed form. For example, if we have a recurrence in the complexity language $\mathsf{rec}(\mathsf{x},\mathsf{Z} \mapsto 2,\mathsf{S} \mapsto \langle n, r\rangle.2{+}r)$ where r is the result of the recursive call, we want to be able to use the _$\leqslant$s_ judgement to prove that $\mathsf{rec}(\mathsf{x},\mathsf{Z} \mapsto 2,\mathsf{S} \mapsto \langle n, r\rangle.2{+}r) \leqslant\mathsf{s}\ \ldots\ \leqslant\mathsf{s}\ 2(n) + 2$. The current rules of the abstract preorder judgement are only helpful in proving the bounding theorem presented in Chapter 3, but for future work, we

```
data CTp : Set where
  ...
  C : CTp
  rnat : CTp
data _|-_ : Ctx → CTp → Set where
  ...
  0 C : ∀ {Γ} → Γ |- C
  1 C : ∀ {Γ} → Γ |- C
  plusC : ∀ {Γ} → Γ |- C → Γ |- C → Γ |- C
  ...
  rz : ∀ {Γ} → Γ |- rnat
  rsuc : ∀ {Γ} → Γ |- rnat → Γ |- rnat
  rrec : ∀ {Γ τ} → Γ |- rnat → (Z' : Γ |- τ)
    → (S' : Γ |- (rnat ->c (τ ->c τ)))
    → (P : Z' ⩽s (app (app S' rz) Z')) → Γ |- τ
  ...
  letc : ∀ {Γ ρ τ} → (ρ :: Γ) |- τ → Γ |- ρ → Γ |- τ
data CTpM : CTp → Set where
  runit : CTpM unit
  rnat-max : CTpM rnat
  _×cm_ : ∀ {τ1 τ2} → CTpM τ1 → CTpM τ2 → CTpM (τ1 ×c τ2)
  _->cm_ : ∀ {τ1 τ2} → CTpM τ2 → CTpM (τ1 ->c τ2)
```

FIGURE 4. Complexity language types, terms, and max types

will add rules to the abstract preorder judgement which will enable proof search in the complexity language and allow us to solve recurrences as intended. Part of the complexity language semantics is given in Figure 5.

```
data _⩽s_ where
  refl-s : ∀ {Γ T} → {e : Γ |- T} → e ⩽s e
  trans-s : ∀ {Γ T} → {e e' e'' : Γ |- T} → e ⩽s e' → e' ⩽s e'' → e ⩽s e''
  cong-refl : ∀ {Γ τ} {e e' : Γ |- τ} → e ≡ e' → e ⩽s e'
  +-unit-l : ∀ {Γ} {e : Γ |- C} → (plusC 0 C e) ⩽s e
  refl-+ : ∀ {Γ} {e0 e1 : Γ |- C} → (plusC e0 e1) ⩽s (plusC e1 e0)
  cong-lproj : ∀ {Γ τ τ'} {e e' : Γ |- (τ ×c τ')} → e ⩽s e' → (l-proj e) ⩽s (l-proj e')
  cong-rproj : ∀ {Γ τ τ'} {e e' : Γ |- (τ ×c τ')} → e ⩽s e' → (r-proj e) ⩽s (r-proj e')
  cong-app : ∀ {Γ τ τ'} {e e' : Γ |- (τ ->c τ')} {e1 : Γ |- τ} → e ⩽s e' → (app e e1) ⩽s (app e' e1)
  ren-cong : ∀ {Γ Γ' τ} {e1 e2 : Γ' |- τ} {ρ : rctx Γ Γ'} → e1 ⩽s e2 → (ren e1 ρ) ⩽s (ren e2 ρ)
  subst-cong : ∀ {Γ Γ' τ} {e1 e2 : Γ' |- τ} {Θ : sctx Γ Γ'} → e1 ⩽s e2 → (subst e1 Θ) ⩽s (subst e2 Θ)
  cong-rec : ∀ {Γ τ} {e e' : Γ |- nat} {e0 : Γ |- τ} {e1 : (nat :: (τ :: Γ)) |- τ}
    → e ⩽s e' → rec e e0 e1 ⩽s rec e' e0 e1
  cong-listrec : ∀ {Γ τ τ'} {e e' : Γ |- list τ} {e0 : Γ |- τ'} {e1 : (τ :: (list τ :: (τ' :: Γ))) |- τ'}
        → e ⩽s e' → listrec e e0 e1 ⩽s listrec e' e0 e1
  lam-s : ∀ {Γ T T'} → {e : (T :: Γ) |- T'} → {e2 : Γ |- T} → subst e (q e2) ⩽s app (lam e) e2
  l-proj-s : ∀ {Γ T1 T2} → {e1 : Γ |- T1} {e2 : Γ |- T2} → e1 ⩽s (l-proj (prod e1 e2))
  r-proj-s : ∀ {Γ T1 T2} → {e1 : Γ |- T1} → {e2 : Γ |- T2} → e2 ⩽s (r-proj (prod e1 e2))
  rec-steps-z : ∀ {Γ T} → {e0 : Γ |- T} → {e1 : (nat :: (T :: Γ)) |- T} → e0 ⩽s (rec z e0 e1)
  rec-steps-s : ∀ {Γ T} → {e : Γ |- nat} → {e0 : Γ |- T} → {e1 : (nat :: (T :: Γ)) |- T}
    → subst e1 (lem4 e (rec e e0 e1)) ⩽s (rec (s e) e0 e1)
  subst-id-l : ∀ {Γ τ} → (e : Γ |- τ) → e ⩽s subst e ids
  subst-rs-l : ∀ {A B C τ} → (ρ : rctx C A) (Θ : sctx A B) (e : B |- τ)
        → ren (subst e Θ) ρ ⩽s subst e (ρ rs Θ)
  subst-compose-l : ∀ {Γ Γ' τ τ1} (Θ : sctx Γ Γ') (v : Γ |- τ) (e : (τ :: Γ' |- τ1))
    → subst (subst e (s-extend Θ)) (q v) ⩽s subst e (lem3' Θ v)
```

FIGURE 5. Complexity language semantics (partial implementation)

CHAPTER 3

Source to Complexity

We define a translation function $||\cdot||$, a monadic transformation from the source language to the complexity language which makes source costs explicit and is the recurrence extraction mechanism for our system. The results obtained by the translation are an upper bound on the costs specified by the source language operational semantics. This is a basic notion of correctness which ensures that the translation from source to complexity does not alter cost information in an undesirable way. We also present two examples of applying the translation function show how our system extracts recurrences from source programs.

1. Translation

The translation of a source term e returns a pair of type $\mathsf{C} \times \langle\langle \tau \rangle\rangle$. The first component of the pair captures the upper bound on the cost of evaluating a source expression, and the second component the upper bound on the *potential*, or size of the value to which e evaluates. Later, we write e_c and e_p to mean the cost and potential components of a complexity expression $e : \mathsf{C} \times \langle\langle \tau \rangle\rangle$. We write $n +_c e$ to mean $(n +_c e_c, e_p)$ for a cost n and complexity expression e.

In Agda, we define type translation and potential mutually:

```
mutual
  ||_|| : Tp → CTp
  ||τ|| = C ×c ⟨⟨ τ ⟩⟩
  ⟨⟨_⟩⟩ : Tp → CTp
  ⟨⟨ unit ⟩⟩ = unit
  ⟨⟨ nat ⟩⟩ = nat
  ⟨⟨ susp A ⟩⟩ = ||A||
  ⟨⟨ A ->s B ⟩⟩ = ⟨⟨ A ⟩⟩ ->c ||B||
```

$$\langle\langle\, A \times_s B \,\rangle\rangle = \langle\langle\, A \,\rangle\rangle \times_c \langle\langle\, B \,\rangle\rangle$$

$$\langle\langle\, \mathsf{list}\ A \,\rangle\rangle = \mathsf{list}\ \langle\langle\, A \,\rangle\rangle$$

$$\langle\langle\, \mathsf{bool} \,\rangle\rangle = \mathsf{bool}$$

Translation of costs and contexts is also straightforward:

$$\mathsf{interp\text{-}Cost} : \forall\, \{\Gamma\} \to \mathsf{Cost} \to \Gamma\ \mathsf{Complexity.}\vdash C$$

$$\mathsf{interp\text{-}Cost}\ 0\ c = 0\ C$$

$$\mathsf{interp\text{-}Cost}\ 1\ c = 1\ C$$

$$\mathsf{interp\text{-}Cost}\ (m +_c n) = \mathsf{plusC}\ (\mathsf{interp\text{-}Cost}\ m)\ (\mathsf{interp\text{-}Cost}\ n)$$

$$\langle\langle \_ \rangle\rangle_c : \mathsf{Source.Ctx} \to \mathsf{Complexity.Ctx}$$

$$\langle\langle\, [\,] \,\rangle\rangle_c = [\,]$$

$$\langle\langle\, x :: \Gamma \,\rangle\rangle_c = \langle\langle\, x \,\rangle\rangle :: \langle\langle\, \Gamma \,\rangle\rangle_c$$

The translation of terms is given in Figure 1, and is a direct implementation of the translation given in [DLR15].

2. Bounding

It is important that the monadic transformation from source to complexity extracts cost information reliably and does not alter cost information in an undesirable way. We do this by proving a bounding theorem that ensures the costs assigned to source programs by the translation function are upper bounds on their running time specified by the source language operational semantics. We rely on a logical relation[1] between the source and complexity semantics to prove this theorem. We now introduce some necessary definitions and lemmas for the bounding theorem.

DEFINITION 3.1 (Bounding relation).

(1) *Let e be a closed source language expression and E a closed complexity expression. We write $e \sqsubseteq_\tau E$ to mean: if $e \downarrow^n v$, then*

[1] See [Sta85] and [Plo73] for an introduction to logical relations.

```
||_||e : ∀ {Γ τ} → Γ Source.|- τ → ⟨⟨ Γ ⟩⟩c Complexity.|- ||τ||

||unit||e = prod 0 C unit

||var x||e = prod 0 C (var (lookup x))

||z||e = prod 0 C z

||suc e||e = prod (l-proj (||e||e)) (s (r-proj (||e||e)))

||rec e e0 e1||e = (l-proj (||e||e)) +C (rec (r-proj ||e||e) (1 C +C ||e0||e) (1 C +C ||e1||e))

||lam e||e = prod 0 C (lam ||e||e)

||app e1 e2||e =

  prod (plusC (plusC (plusC 1 C (l-proj ||e1||e)) (l-proj ||e2||e))

    (l-proj (app (r-proj ||e1||e) (r-proj ||e2||e))))

  (r-proj (app (r-proj ||e1||e) (r-proj ||e2||e)))

||prod e1 e2||e = prod (plusC (l-proj (||e1||e)) (l-proj (||e2||e))) (prod (r-proj (||e1||e)) (r-proj (||e2||e)))

||delay e||e = prod 0 C (||e||e)

||force e||e = prod (plusC (l-proj (||e||e)) (l-proj (r-proj ||e||e))) (r-proj (r-proj (||e||e)))

||split e0 e1||e = prod (plusC (l-proj (||e0||e)) (l-proj E1)) (r-proj E1)

  where E1 = (Complexity.subst ||e1||e (Complexity.lem4 (l-proj (r-proj ||e0||e)) (r-proj (r-proj ||e0||e))))

||nil||e = prod 0 C nil

||e ::s e₁||e = prod (plusC (l-proj ||e||e) (l-proj ||e₁||e)) ((r-proj ||e||e) ::c (r-proj ||e₁||e))

||listrec e e₁ e₂||e = l-proj ||e||e +C listrec (r-proj ||e||e) (1 C +C ||e₁||e) (1 C +C ||e₂||e)

||true||e = prod 0 C true

||false||e = prod 0 C false
```

FIGURE 1. Direct implementation of the translation function $||\cdot||$ from [DLR15]

(a) $n \leqslant E_c$; *and*

(b) $v \sqsubseteq^{val}_{\tau} E_p$.

(2) *Let v be a source language value and E a closed complexity expression. We write $v \sqsubseteq^{val}_{\tau} E$ to mean:*

(a) $Z \sqsubseteq^{val}_{nat} E$ *if* $\mathsf{Z} \leqslant E$

(b) *(*$\mathsf{S}\ n$*)* $\sqsubseteq^{val}_{nat} E$ *if there is an* E' *such that* $n \sqsubseteq^{val}_{nat} E'$ *and (*$\mathsf{S}\ E$*')* $\leqslant E$

(c) $\mathsf{nil} \sqsubseteq^{val}_{list\tau} E$ *if* $\mathsf{nil} \leqslant E$

(d) $x ::\mathsf{s}\ xs \sqsubseteq^{val}_{list\tau} E$ *if there are* E', E'' *such that* $x \sqsubseteq^{val}_{\tau} E'$, $xs \sqsubseteq^{val}_{list\tau} E''$, *and* $E' ::\mathsf{c}\ E'' \leqslant E$.

(e) $\langle v0, v1\rangle \sqsubseteq^{val}_{\tau_0\times\tau_1} \langle E0, E1\rangle$ *if* $v_i \sqsubseteq^{val}_{\tau_i} E_i$ *for i = 0, 1.*

(f) $\mathsf{delay}(e) \sqsubseteq^{val}_{susp\tau} E$ *if* $e \sqsubseteq_\tau E$.

(g) $\lambda x.e \sqsubseteq^{val}_{\sigma\to\tau} E$ *if whenever* $v \sqsubseteq^{val}_{\sigma} E'$, $e[v/x] \sqsubseteq_\tau E(E')$

We write $e \sqsubseteq_\tau E$ to mean "e is *expression-bounded* by E" and $v \sqsubseteq^{val}_{\tau} E$ to mean "v is *value-bounded* by E." Figure 2 shows how the Agda implementation easily corresponds to the paper definition of the bounding relation.

LEMMA 3.2 (Weakening).

(1) *If* $e \sqsubseteq_\tau E$ *and* $E \leqslant_{|\tau|} E$*' then* $e \sqsubseteq_\tau E$*'.*

(2) *If* $v \sqsubseteq^{val}_{\tau} E$ *and* $E \leqslant_{\langle\langle\tau\rangle\rangle} E$*' then* $e \sqsubseteq^{val}_{\tau} E'$

PROOF. The proof is 20 lines of Agda. For (1), suppose $e \sqsubseteq_\tau E$ and $E \leqslant_{||\tau||} E'$. By definition, we have $e \sqsubseteq_\tau E = (e_c \leqslant_{\mathbb{C}} E_c, e_p \sqsubseteq^{val}_{\tau} E_p)$ and $E \leqslant_{||\tau||} E' = E \leqslant_{\mathbb{C}\times\langle\langle\tau\rangle\rangle} E'$, so it suffices to show $e_c \leqslant_{\mathbb{C}} E'_c$ and $e_p \sqsubseteq^{val}_{\tau} E'_p$. These are both true by transitivity and congruence of products on $E \leqslant_{\mathbb{C}\times\langle\langle\tau\rangle\rangle} E'$. (2) is proved by induction on the type of v. All cases are either immediate by transitivity, or the induction hypothesis (1). □

LEMMA 3.3 (Inversion). *If* $v \downarrow^n v$*', then* $n \leqslant 0$ *and* $v = v$*'.*

PROOF. The proof is 20 lines of Agda. □

THEOREM 3.4 (Bounding Theorem). *If* $\Gamma \vdash e : \tau$, θ *is a substitution of all variables in a source context, and* Θ *is a corresponding substitution of all variables in a complexity context, then* $e[\theta] \sqsubseteq_\tau ||e||[\Theta]$.

```
mutual
  valBound : ∀ {τ} → (e : [] Source.|- τ) → val e → [] Complexity.|- ⟨⟨ τ ⟩⟩ → Set
  valBound z z-isval E = z ⩽s E
  valBound (suc e) (suc-isval e v) E = Σ (λ E' → valBound e v E' × s E' ⩽s E)
  valBound (prod e1 e2) (pair-isval e1 e2 v1 v2) E =
    valBound e1 v1 (l-proj E) × valBound e2 v2 (r-proj E)
  valBound {τ1 ->s τ2} (lam e) (lam-isval e) E =
    (v₁ : [] Source.|- τ1) (vv : val v₁) (E1 : [] Complexity.|- ⟨⟨ τ1 ⟩⟩) →
    valBound v₁ vv E1 →
    expBound (Source.subst e (Source.q v₁)) (app E E1)
  valBound unit unit-isval E = Unit
  valBound (delay e) (delay-isval e) E = expBound e E
  valBound nil nil-isval E = nil ⩽s E
  valBound (x ::s xs) (cons-isval x xs v v₁) E =
    Σ (λ E' → Σ (λ E'' → (valBound x v E' × valBound xs v₁ E'') × (E' ::c E'') ⩽s E))
  valBound true true-isval E = true ⩽s E
  valBound false false-isval E = false ⩽s E
  expBound : ∀ {τ} → [] Source.|- τ → [] Complexity.|- ∨ τ ∨ → Set
  expBound {τ} e b = (v₁ : [] Source.|- τ) (vv : val v₁) (n : Cost)
    → evals e v₁ n → interp-Cost n ⩽s l-proj b × valBound v₁ vv (r-proj b)
```

FIGURE 2. Agda implementation of the bounding relation.

PROOF. The proof is by induction on $\Gamma \vdash e : \tau$, and is 150 lines of Agda. A more detailed account of the proof of bounding can be found in [Hud15]. □

3. Examples

In this section, we demonstrate how the translation function yields recurrences with two example programs. As a first example, consider a function which appends two lists:

$\mathsf{append} : \{\mathsf{A} : \mathsf{Set}\}\ (\mathsf{l1}\ \mathsf{l2} : \mathsf{List\ A}) \to \mathsf{List\ A}$

$\mathsf{append}\ [\,]\ \mathsf{ys} = \mathsf{ys}$

$\mathsf{append}\ (\mathsf{x} :: \mathsf{xs})\ \mathsf{ys} = \mathsf{x} :: \mathsf{append\ xs\ ys}$

We can implement this in the source language as follows:

$$\mathsf{append} := \lambda l1.\lambda l2.\mathsf{rec}(l1, \mathsf{Nil} \mapsto l2 \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.\mathsf{Cons}(x, \mathsf{force}\ r))$$

where r is the suspended result of the recursive call on xs. Following the translation, we obtain

$$\begin{aligned}
||\ \mathsf{append}\ || =& ||\ \lambda l1.\lambda l2.\mathsf{rec}(l1, \mathsf{Nil} \mapsto l2 \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.\mathsf{Cons}(x, \mathsf{force}\ r))\ || \\
&= (0, \lambda l1.\ ||\ \lambda l2.\mathsf{rec}(l1, \mathsf{Nil} \mapsto l2 \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.\mathsf{Cons}(x, \mathsf{force}\ r))\ ||) \\
&\text{(by definition of } ||\cdot||) \\
&= (0, \lambda l1.(0, \lambda l2.\ ||\ \mathsf{rec}(l1, \mathsf{Nil} \mapsto l2 \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.\mathsf{Cons}(x, \mathsf{force}\ r))\ ||)) \\
&\text{(by definition of } ||\cdot||)
\end{aligned}$$

At this point, we would like to evaluate the translation of the recursor separately for readability:

$$\begin{aligned}
&||\ \mathsf{rec}(l1, \mathsf{Nil} \mapsto l2 \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.\mathsf{Cons}(x, \mathsf{force}\ r))\ || \\
=&||\ l1\ ||_c +_c \mathsf{rec}(||\ l1\ ||_p, \mathsf{Nil} \mapsto 1 +_c ||\ l2\ || \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.1 +_c ||\ \mathsf{Cons}(x, \mathsf{force}\ r)\ ||) \\
&\text{(by definition of } ||\cdot||) \\
=& \mathsf{rec}(l1, \mathsf{Nil} \mapsto 1 +_c (0, l2) \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.(1 + r_c, \mathsf{Cons}(x, r_p))) \\
&\text{(by definition of } ||\ l1\ ||, ||\ l2\ ||, ||\ \mathsf{force}\ r\ ||) \\
=& \mathsf{rec}(l1, \mathsf{Nil} \mapsto (1, l2) \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.(1 + r_c, \mathsf{Cons}(x, r_p))) \\
&\text{(by definition of } +_c)
\end{aligned}$$

Plugging this back into the original equation, we obtain

$$(0, \lambda l1.(0, \lambda l2.\mathsf{rec}(l1, \mathsf{Nil} \mapsto (1, l2) \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.(1 + r_c, \mathsf{Cons}(x, r_p)))))$$

A manual analysis of append gives us a recurrence of the form

$$T_{\mathsf{append}}(0) = c_0$$
$$T_{\mathsf{append}}(n) = c_1 + T_{\mathsf{append}}(n-1) \text{ for non-zero } n$$

where c_0, c_1 are fixed integer constants, and n is the size of the first list argument. The size of the second list argument is irrelevant because the function does not depend on its length. In addition, observe that the cost component of the resulting recurrence yields a result that is similar to the above:

$$|| \ \mathsf{append}(\mathsf{Nil})(ys) \ ||_c = 1$$
$$|| \ \mathsf{append}(\mathsf{Cons}(x,xs))(ys) \ ||_c = 1 + || \ \mathsf{append}(xs) \ ||_c$$

Now consider a function which uses append, like a function to reverse a list:

```
rev : {A : Set} (l1 : List A) → List A
rev [] = []
rev (x :: xs) = append (rev xs) (x :: [])
```

We can implement this in the source language as follows:

$$\mathsf{rev} := \lambda l.\mathsf{rec}(l, \mathsf{Nil} \mapsto \mathsf{Nil} \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.\mathsf{append}(\mathsf{force}\ r, \mathsf{Cons}(x, \mathsf{Nil})))$$

where r is the suspended result of the recursive call on xs. Following the translation, we obtain

$$\begin{aligned} || \ \mathsf{rev} \ || &= || \ \lambda l.\mathsf{rec}(l, \mathsf{Nil} \mapsto \mathsf{Nil} \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.\mathsf{append}(\mathsf{force}\ r, \mathsf{Cons}(x, \mathsf{Nil}))) \ || \\ &= (0, \lambda l.(|| \ \mathsf{rec}(l, \mathsf{Nil} \mapsto \mathsf{Nil} \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.\mathsf{append}(\mathsf{force}\ r, \mathsf{Cons}(x, \mathsf{Nil}))) \ ||)) \\ &\quad \text{(by definition of } || \cdot ||) \end{aligned}$$

At this point again, we would like to evaluate the recursor separately for readability:

$$|| \ \mathsf{rec}(l, \mathsf{Nil} \mapsto \mathsf{Nil} \mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.\mathsf{append}(\mathsf{force}\ r, \mathsf{Cons}(x, \mathsf{Nil}))) \ ||$$
$$= || \ l \ ||_c +_c \mathsf{rec}(|| \ l \ ||_p, \mathsf{Nil} \mapsto 1 +_c || \ \mathsf{Nil} \ ||| \ \mathsf{Cons} \mapsto \langle x, \langle x, xs\rangle\rangle.1 +_c || \ \mathsf{append}(\mathsf{force}\ r, \mathsf{Cons}(x, \mathsf{Nil})) \ ||)$$

(by definition of $|| \cdot ||$)

The case for Nil is straightforward. We focus on the application in the recursive branch instead, recalling $||\ e_0\ e_1\ ||= (1+||\ e_0\ ||_c + ||\ e_1\ ||_c)+_c ||\ e_0\ ||_p||\ e_1\ ||_p$ for source terms e_0, e_1:

$$
\begin{aligned}
&||\ (\mathsf{append}(\mathsf{force}\ r))\mathsf{Cons}(x,\mathsf{Nil})\ || \\
&= (1+||\ \mathsf{append}(\mathsf{force}\ r)\ ||_c + ||\ \mathsf{Cons}(x,\mathsf{Nil})\ ||_c)+_c ||\ \mathsf{append}(\mathsf{force}\ r)\ ||_p||\ \mathsf{Cons}(x,\mathsf{Nil})\ ||_p \\
&\text{(by definition of } ||\cdot||) \\
&= (1+||\ \mathsf{append}(\mathsf{force}\ r)\ ||_c)+_c ||\ \mathsf{append}(\mathsf{force}\ r)\ ||_p\ \mathsf{Cons}(x,\mathsf{Nil}) \\
&\text{(by definition, ||Cons(}x\text{,Nil)|| = (0 , Cons(}x\text{,Nil)))}
\end{aligned}
$$

A separate analysis of the inner application $||\ \mathsf{append}(\mathsf{force}\ r)\ ||$ yields

$$
\begin{aligned}
||\ \mathsf{append}(\mathsf{force}\ r)\ || &= (1+||\ \mathsf{append}\ ||_c + r_c)+_c ||\ \mathsf{append}\ ||_p\ \ r_p && \text{(by definition of } ||\cdot||) \\
&= (1+r_c+(||\ \mathsf{append}\ ||_p\ \ r_p)_c, (||\ \mathsf{append}\ ||_p\ \ r_p)_p) && \text{(by definition of } +_c)
\end{aligned}
$$

When we place the expanded term back where we left off, we obtain

$$
\begin{aligned}
&(1+(1+r_c+(||\ \mathsf{append}\ ||_p\ \ r_p)_c))+_c (||\ \mathsf{append}\ ||_p\ \ r_p)_p\ \mathsf{Cons}(x,\mathsf{Nil}) \\
&= (2+r_c+(||\ \mathsf{append}\ ||_p\ \ r_p)_c)+_c (||\ \mathsf{append}\ ||_p\ \ r_p)_p\ \mathsf{Cons}(x,\mathsf{Nil}) \\
&= (2+r_c+(||\ \mathsf{append}\ ||_p\ \ r_p)_c + (||\ \mathsf{append}\ ||_p\ \ r_p)_p\ \mathsf{Cons}(x,\mathsf{Nil})_c, (||\ \mathsf{append}\ ||_p\ \ r_p)_p\ \mathsf{Cons}(x,\mathsf{Nil})) \\
&\text{(by definition of } +_c)
\end{aligned}
$$

Adding the cost of the recursive branch and placing the recursor back into the original term, the complete translation for rev is as follows:

$$
\begin{aligned}
&(0, \lambda l.(\mathsf{rec}(l, \mathsf{Nil} \mapsto (1,\mathsf{Nil}) \\
&\mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.(3+r_c+(||\ \mathsf{append}\ ||_p\ \ r_p)_c + ((||\ \mathsf{append}\ ||_p\ \ r_p)_p\ \mathsf{Cons}(x,\mathsf{Nil}))_c, \\
&(||\ \mathsf{append}\ ||_p\ \ r_p)_p\ \mathsf{Cons}(x,\mathsf{Nil}))
\end{aligned}
$$

We can reduce this to

$$(0, \lambda l.(\mathsf{rec}(l, \mathsf{Nil} \mapsto (1, \mathsf{Nil})$$
$$\mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.(3 + r_c + ((|| \ \mathsf{append} \ ||_p \ \ r_p)_p \ \mathsf{Cons}(x, \mathsf{Nil}))_c,$$
$$(|| \ \mathsf{append} \ ||_p \ \ r_p)_p \ \mathsf{Cons}(x, \mathsf{Nil}))$$

because the cost of a partial application $|| \ \mathsf{append} \ ||_p \ \ r_p$ is 0. In addition, observe that the cost component of the extracted term resembles the recurrence for rev generated by hand:

$$T_{\mathsf{rev}}(0) = c_0$$
$$T_{\mathsf{rev}}(n) = c_1 + T_{\mathsf{append}}(n-1) + T_{\mathsf{rev}}(n-1) \text{ for non-zero } n$$

4. Optimization

The translation from source to complexity yields precise bounds on the running time of source programs, but the implementation given earlier this chapter is problematic in that it duplicates many recursive calls. This expands term size exponentially within Agda and consequently reduces the speed at which our system extracts recurrences.

To avoid this exponential increase in term size during translation, we can implement the translation function using let bindings to reduce the number of recursive calls we have to make. Now, for example, instead of the original implementation of translation for complexity pairs

```
||prod e1 e2||e =
  prod (plusC (l-proj (||e1||e)) (l-proj (||e2||e)))
  (prod (r-proj (||e1||e)) (r-proj (||e2||e)))
```

which appeals to 4 recursive calls, we have

```
||prod e1 e2||e =
  letc (letc (prod (plusc (l-proj (var (is i0))) (l-proj (var i0)))
  (prod (r-proj (var (is i0))) (r-proj (var i0)))) (complexity.wkn ||e2||e)) ||e1||e
```

which has only 2 recursive calls, and where letc (equivalent to app (lam e) e') is defined as

```
letc : ∀ {Γ ρ τ} → (ρ :: Γ) |- τ → Γ |- ρ → Γ |- τ
```

Using the previous implementation, the translation of a simple function to double a natural number yields a term that is more than 10,000 lines long. However, with the optimized translation function (which is named $||\cdot||_{\mathsf{e}'}$ in the code) which uses the new letc construct, the translation of the same function consists of only 14 lines.

Although we prove the bounding theorem with respect to the direct implementation of the translation function, we conjecture that the bounding property would not be affected if we proved the bounding theorem with respect to the optimized implementation – this is due to the fact that $||\cdot||_{\mathsf{e}}$ and $||\cdot||_{\mathsf{e}'}$ only differ in the use of let bindings.

CHAPTER 4

Complexity to Preorders

In this chapter, we give a denotational semantics of the complexity language which interprets types as preorders and terms as monotone maps between preorders. We also prove that the complexity language is sound with respect to the denotational semantics. In the proceeding section, we introduce all necessary mathematical prerequisites to the reader.

1. Preorders

DEFINITION 4.1 (Preorder). *A* preorder *is a pair* $(A, \leqslant)$ *consisting of a set A, and a binary relation on A,* $\leqslant$*, which is reflexive and transitive.*

We write $\leqslant_A$ to mean the $\leqslant$ relation defined on A, and call A the *underlying set* of a preorder $(A, \leqslant)$. Preorders can be defined in Agda as a record type:

```
record Preorder-str (A : Set) : Set1 where
  constructor preorder
  field
    ⩽ : A → A → Set
    refl : ∀ x → ⩽ x x
    trans : ∀ x y z → ⩽ x y → ⩽ y z → ⩽ x z
```

In Agda, the preorder over $\mathbb{N}$ is defined as follows:

```
nat-ispreorder : Preorder-str Nat
nat-ispreorder = preorder ⩽nat nat-refl nat-trans
```

where Nat is the built-in implementation of $\mathbb{N}$, $\leqslant$nat is the usual $\leqslant$ relation defined on $\mathbb{N}$, and nat-refl and nat-trans are proofs that $\leqslant$nat is reflexive and transitive. Preorders over cartesian products and lists are also defined similarly. Later, we write PREORDER to mean the pair of a

set together with the proof that there is a preorder over that set, i.e. PREORDER = Σ (λ (A : Set) → Preorder-str A).

As discussed in the previous chapter, for future work, we need preorders with a notion of maximums so we can interpret complexity max types appropriately:

```
record Preorder-max-str {A : Set} (PA : Preorder-str A) : Set where
  constructor preorder-max
  field
    max : A → A → A
    max-l : ∀ l r → Preorder-str._⩽ PA l (max l r)
    max-r : ∀ l r → Preorder-str._⩽ PA r (max l r)
    max-lub : ∀ k l r → Preorder-str._⩽ PA l k → Preorder-str._⩽ PA r k
      → Preorder-str._⩽ PA (max l r) k
```

That is, a preorder with maximums is a preorder equipped with an operation to compute the maximum of two given values, and proofs that the operation preserves the ordering on terms as well as lower upper bounds.

DEFINITION 4.2 (Monotone function). *Given preorders* $(A, \leqslant_A)$ *and* $(B, \leqslant_B)$*, a function* $f : A \to B$ *is* monotone *if* $\forall x, y \in A$*, if* $x \leqslant_A y$*, then* $f(x) \leqslant_B f(y)$.

Monotone functions can also be defined in Agda as a record type, whose fields are the function itself and the proof that the function is monotone:

```
record Monotone (A : Set) (B : Set) (PA : Preorder-str A) (PB : Preorder-str B) : Set where
  constructor monotone
  field
    f : A → B
    is-monotone : ∀ (x y : A) → Preorder-str._⩽ PA x y → Preorder-str._⩽ PB (f x) (f y)
```

For brevity, we write MONOTONE PA PB to mean the monotone function from a preorder PA to another preorder PB:

```
MONOTONE : (PΓ PA : PREORDER) → Set
MONOTONE (Γ, PΓ) (A, PA) = Monotone Γ A PΓ PA
```

The preorder on monotone functions between preorders is defined such that the order on monotone functions is simply the pointwise order on the underlying functions. More precisely, given preorders $(A, \leqslant_A)$ and $(B, \leqslant_B)$ and monotone functions $f : A \to B$ and $g : A \to B$, $f \leqslant_{A \to B} g$ means $\forall x : A, f(x) \leqslant_B g(x)$.

Given the formation rules for each type in terms of preorders, we also need to know how to introduce and eliminate terms characterized as monotone maps between preorders. As a concrete example, consider cartesian products.

Suppose we are given preorders $(\Gamma, \leqslant_\Gamma), (A, \leqslant_A), (B, \leqslant_B)$ with monotone maps $f : \Gamma \to A$ and $g : \Gamma \to B$. Define $p : \Gamma \to A \times B$ by $p(x) = (f(x), g(x))$ for $x \in \Gamma$. To conclude that we can introduce pairs as monotone maps between preorders, we need to show that p is monotonic.

THEOREM 4.3. *$\forall a, b \in \Gamma$, if $a \leqslant_\Gamma b$, then $p(a) \leqslant_{A \times B} p(b)$.*

PROOF. Assume we have $a, b \in \Gamma$, such that $a \leqslant_\Gamma b$. By definition, $p(a) = (f(a), g(a))$, and $p(b) = (f(b), g(b))$. Recall $(f(a), g(a)) \leqslant_{A \times B} (f(b), g(b))$ if and only if $f(a) \leqslant_A f(b)$ and $g(a) \leqslant_B g(b)$. However, these are both true by monotonicity of f and g, so we can conclude p is monotonic as well. It follows immediately that we can introduce product types in our preorder interpretation. □

The proofs that the projections $\pi_1 p$ and $\pi_2 p$ for a product p are monotone are similar. In Agda, the complete characterization of introduction and elimination forms for product types as monotonic functions between preorders is as follows:

```
pair' : ∀ {PΓ PA PB} → MONOTONE PΓ PA → MONOTONE PΓ PB
  → MONOTONE PΓ (PA ×p PB)
pair' (monotone f f-ismono) (monotone g g-ismono) =
  monotone (λ x → f x, g x) (λ x y z → f-ismono x y z, g-ismono x y z)
fst' : ∀ {PΓ PA PB} → MONOTONE PΓ (PA ×p PB) → MONOTONE PΓ PA
```

```
fst' (monotone f f-ismono) = monotone (λ x → fst (f x)) (λ x y z → fst (f-ismono x y z))

snd' : ∀ {PΓ PA PB} → MONOTONE PΓ (PA ×p PB) → MONOTONE PΓ PB
snd' (monotone f f-ismono) = monotone (λ x → snd (f x)) (λ x y z → snd (f-ismono x y z))
```

where PA ×p PB is the preorder formed by the cartesian product of preorders PA and PB. The corresponding rules for abstractions, function applications, natural numbers, and lists are defined similarly. Monotonicity is also preserved by the identity function and composition of monotone functions:

```
id : ∀ {Γ} → MONOTONE Γ Γ
id = λ {Γ} → monotone (λ x → x) (λ x y x₁ → x₁)

comp : ∀ {PA PB PC} → MONOTONE PA PB → MONOTONE PB PC
  → MONOTONE PA PC
comp (monotone f f-ismono) (monotone g g-ismono) =
  monotone (λ x → g (f x)) (λ x y x₁ → g-ismono (f x) (f y) (f-ismono x y x₁))
```

2. Interpretation of Complexity Types

We choose to interpret complexity types as preorders, rather than just domains. The function $[\cdot]$t maps complexity types to preorders:

```
[_] t : CTp → PREORDER
[unit] t = unit-p
[nat] t = Nat, ♭nat-p
[τ ->c τ₁] t = [τ] t ->p [τ₁] t
[τ ×c τ₁] t = [τ] t ×p [τ₁] t
[list τ] t = (List (fst [τ] t)), list-p (snd [τ] t)
[bool] t = Bool, bool-p
[C] t = Nat, nat-p
[rnat] t = Nat, nat-p
```

where A ->p B is the preorder formed by the monotone function between preorders A and B. Later, given a preorder PA = $(A, \leqslant_A)$, we write el PA to mean the underlying set of the preorder; i.e. el PA = A. We have a separate function $[\cdot]$tm which maps complexity max types to preorders with maximums:

```
[_] tm : ∀ {A} → CTpM A → Preorder-max-str (snd [A] t)
[runit] tm = unit-pM
[rn] tm = nat-pM
[e ×cm e₁] tm = axb-pM [e] tm [e₁] tm
[_->cm_ {τ1} e] tm = mono-pM [e] tm
```

where axb-pM is the preorder with maximums formed by the cartesian product of [e]tm and [e_1]tm, and mono-pM is the preorder formed by the monotone function whose codomain is [e]tm.

3. Interpretation of Complexity Terms

Given a complexity term $\Gamma \vdash \tau$, we define a function interpE which interprets complexity terms as monotone maps between preorders:

$$\mathsf{interpE}(\Gamma \vdash \tau) = \mathsf{el}([\Gamma]\mathsf{c}\text{->}\mathsf{p}[\tau]\mathsf{t})$$

where the denotation of a typing context, $[\Gamma]$c, is defined by induction on the size of Γ:

```
[_] c : Ctx → PREORDER
[[]] c = unit-p
[τ :: Γ] c = [Γ] c ×p [τ] t
```

Define the lookup function for the preorder semantics by induction on the index of the variable in the context:

```
lookup : ∀ {Γ τ} → τ ∈ Γ → el ([Γ] c ->p [τ] t)
lookup (i0 {Γ} {τ}) = snd' id
lookup (iS {Γ} {τ} {τ1} x) = comp (fst' id) (lookup x)
```

```
interpE : ∀ {Γ τ} → Γ |- τ → el ([Γ] c ->p [τ] t)
interpE unit = monotone (λ x → <>) (λ x y x₁ → <>)
interpE 0 C = monotone (λ x → Z) (λ x y x₁ → <>)
interpE 1 C = monotone (λ x → S Z) (λ x y x₁ → <>)
interpE (var x) = lookup x
interpE z = monotone (λ x → Z) (λ x y x₁ → <>)
interpE (s e) =
  monotone (λ x → S (Monotone.f (interpE e) x))
  (λ x y x₁ → Monotone.is-monotone (interpE e) x y x₁)
interpE (rec e e₁ e₂) =
  comp (pair' id (interpE e)) (♭rec' (interpE e₁) (interpE e₂))
interpE (lam e) = lam' (interpE e)
interpE (app e e₁) = app' (interpE e) (interpE e₁)
interpE (prod e e₁) = pair' (interpE e) (interpE e₁)
interpE (l-proj e) = fst' (interpE e)
interpE (r-proj e) = snd' (interpE e)
interpE nil = nil'
interpE (e ::c e₁) = cons' (interpE e) (interpE e₁)
interpE (listrec e e₁ e₂) =
  comp (pair' id (interpE e)) (lrec' (interpE e₁) (interpE e₂))
interpE true = monotone (λ x → True) (λ x y x₁ → <>)
interpE false = monotone (λ x → False) (λ x y x₁ → <>)
interpE (letc e e') = app' (lam' (interpE e)) (interpE e')
```

FIGURE 1. Implementation of interpE (partial)

Part of the implementation of interpE is given in Figure 1. Observe that the semantics is compositional; the denotation of a term is determined solely by the denotation of its subterms.

4. Interpretation of Substitutions and Renamings

Define the interpretation of substitutions and renamings by induction on the length of the context which they map to:

```
interpS : ∀ {Γ Γ'} → sctx Γ Γ' → el ([Γ] c ->p [Γ'] c)
interpS {Γ' = []} Θ = monotone (λ _ → <>) (λ x y x₁ → <>)
interpS {Γ' = τ :: Γ'} Θ =
  monotone
    (λ x → Monotone.f (interpS (throw-s Θ)) x , Monotone.f (interpE (Θ i0)) x)
    (λ x y x₁ → Monotone.is-monotone (interpS (throw-s Θ)) x y x₁ ,
      (Monotone.is-monotone (interpE (Θ i0)) x y x₁))
interpR : ∀ {Γ Γ'} → rctx Γ Γ' → MONOTONE [Γ] c [Γ'] c
interpR {Γ' = []} ρ = monotone (λ _ → <>) (λ x y x₁ → <>)
interpR {Γ' = τ :: Γ'} ρ =
  monotone
    (λ x → (Monotone.f (interpR (throw-r ρ)) x) , (Monotone.f (lookup (ρ i0)) x))
    (λ x y x₁ → (Monotone.is-monotone (interpR (throw-r ρ)) x y x₁) ,
      (Monotone.is-monotone (lookup (ρ i0)) x y x₁))
```

LEMMA 4.4 (Substitution commutes with interpretation). *For all types τ, contexts Γ, Γ', substitutions θ, terms $e : \tau$, and elements $k \in \Gamma$,*

(1) $[\![e[\theta]]\!]k \leqslant_{[\![\tau]\!]} [\![e]\!]([\![\theta]\!]k)$, *and*
(2) $[\![e]\!]([\![\theta]\!]k) \leqslant_{[\![\tau]\!]} [\![e[\theta]]\!]k$.

PROOF. The theorems are stated in Agda as follows:

```
subst-eq-l : ∀ {Γ Γ' τ} → (Θ : sctx Γ Γ') → (e : Γ' |- τ) → (k : fst [Γ] c)
  → Preorder-str.⩽ (snd [τ] t)
    (Monotone.f (interpE (subst e Θ)) k)
    (Monotone.f (interpE e) (Monotone.f (interpS Θ) k))
```

```
subst-eq-r : ∀ {Γ Γ' τ} → (Θ : sctx Γ Γ') → (e : Γ' |- τ) → (k : fst [Γ] c)
  → Preorder-str.⩽ (snd [τ] t)
    (Monotone.f (interpE e) (Monotone.f (interpS Θ) k))
    (Monotone.f (interpE (subst e Θ)) k)
```

We then proceed by induction on e. The proofs for each direction are about 180 lines of Agda. □

LEMMA 4.5 (Renaming commutes with interpretation). *For all types τ, contexts Γ, Γ', renamings ρ, terms $e : \tau$, and elements $k \in \Gamma$,*

(1) $[\![e[\rho]]\!]k \leqslant_{[\![\tau]\!]} [\![e]\!]([\![\rho]\!]k)$, *and*
(2) $[\![e]\!]([\![\rho]\!]k) \leqslant_{[\![\tau]\!]} [\![e[\rho]]\!]k$.

PROOF. The theorems are stated in Agda as follows:

```
ren-eq-l : ∀ {Γ Γ' τ} → (ρ : rctx Γ Γ') → (e : Γ' |- τ) → (k : fst [Γ] c)
  → Preorder-str.⩽ (snd [τ] t)
    (Monotone.f (interpE (ren e ρ)) k)
    (Monotone.f (interpE e) (Monotone.f (interpR ρ) k))
ren-eq-r : ∀ {Γ Γ' τ} → (ρ : rctx Γ Γ') → (e : Γ' |- τ) → (k : fst [Γ] c)
  → Preorder-str.⩽ (snd [τ] t)
    (Monotone.f (interpE e) (Monotone.f (interpR ρ) k))
    (Monotone.f (interpE (ren e ρ)) k)
```

We proceed by induction again on e. The proofs for each direction are about 170 lines of Agda.

□

In addition to proving that interpretation commutes with substitutions and renamings, we must also prove that interpretation commutes with composition of these operations, like we did for the composition lemmas in Chapter 2.

LEMMA 4.6. *For all contexts Γ, renamings ρ, ρ', and elements $k \in \Gamma$,*

(1) $[\![\rho]\!]([\![\rho']\!]k) \leqslant [\![\rho' \circ \rho]\!]k$, *and*

(2) $[\![\rho' \circ \rho]\!]k \leqslant [\![\rho]\!]([\![\rho']\!]k)$.

LEMMA 4.7. *For all contexts* Γ, *substitutions* θ, *renamings* ρ, *and elements* $k \in \Gamma$,

(1) $[\![\rho]\!]([\![\theta]\!]k) \leqslant [\![\theta \circ \rho]\!]k$, *and*

(2) $[\![\theta \circ \rho]\!]k \leqslant [\![\rho]\!]([\![\theta]\!]k)$.

LEMMA 4.8. *For all contexts* Γ, *substitutions* θ, *renamings* ρ, *and elements* $k \in \Gamma$,

(1) $[\![\theta]\!]([\![\rho]\!]k) \leqslant [\![\rho \circ \theta]\!]k$, *and*

(2) $[\![\rho \circ \theta]\!]k \leqslant [\![\theta]\!]([\![\rho]\!]k)$.

LEMMA 4.9. *For all contexts* Γ, *substitutions* θ, θ', *and elements* $k \in \Gamma$,

(1) $[\![\theta]\!]([\![\theta']\!]k) \leqslant [\![\theta' \circ \theta]\!]k$, *and*

(2) $[\![\theta' \circ \theta]\!]k \leqslant [\![\theta]\!]([\![\theta']\!]k)$.

All of the above lemmas are proved by induction on the length of the context and then applying the IH with Lemmas 4.4 and 4.5.

5. Soundness

THEOREM 4.10 (Soundness). $\forall\ \Gamma \vdash e : \tau$, $\Gamma \vdash e' : \tau$, *and elements* $k \in \Gamma$, *if* $e \leqslant_s e'$ *then* $[\![e]\!]k \leqslant_{[\![\tau]\!]} [\![e']\!]k$.

PROOF. We begin by stating the proof in Agda:

```
sound : ∀ {Γ τ} (e e' : Γ |- τ) → e ≤s e' → PREORDER≤ ([Γ] c ->p [τ] t) (interpE e) (interpE e')
```

The proof is by induction on the abstract preorder judgement of the complexity semantics. There are almost 50 cases to prove, but quite a few of them are proved similarly, so we present the proofs for the most distinct cases. The complete proof is more than 500 lines of Agda, and can be found online.

The simplest cases to prove are those which are immediate just by reflexivity and transitivity of preorders. For example, for the judgement

$$\frac{e \leqslant_{\mathsf{s}} e' \qquad e' \leqslant_{\mathsf{s}} e''}{e \leqslant_{\mathsf{s}} e''}$$

where e, e', e'' all have type τ, we are required to show $[\![e]\!]k \leqslant_{[\![\tau]\!]} [\![e'']\!]k$. This is true transitivity of the preorder $[\![\tau]\!]$ with the IHs $[\![e]\!]k \leqslant_{[\![\tau]\!]} [\![e']\!]k$ and $[\![e']\!]k \leqslant_{[\![\tau]\!]} [\![e'']\!]k$. We can represent this in Agda as

```
sound {Γ} {τ} e e'' (trans-s {.Γ} {.τ} {.e} {e'} {.e''} d d₁) k =
  Preorder-str.trans (snd [τ] t)
    (Monotone.f (interpE e) k)
    (Monotone.f (interpE e') k)
    (Monotone.f (interpE e'') k)
      (sound e e' d k)
      (sound e' e'' d₁ k)
```

where d and d_1 are proofs that e $\leqslant_{\mathsf{s}}$ e' and e' $\leqslant_{\mathsf{s}}$ e'' respectively.

A more interesting case is for the judgement describing congruence of renaming:

$$\frac{e \leqslant_{\mathsf{s}} e'}{e[\rho] \leqslant_{\mathsf{s}} e'[\rho]}$$

where $e, e' : \tau$, and ρ is a renaming. We want to show $[\![e[\rho]]\!]k \leqslant_{[\![\tau]\!]} [\![e'[\rho]]\!]k$, with the IH $[\![e]\!]k \leqslant_{[\![\tau]\!]} [\![e']\!]k$ for any $k \in \Gamma$. We can use the IH if we commute the renaming inwards using Lemma 4.5(1):

$$\begin{aligned}
[\![e[\rho]]\!]k &\leqslant_{[\![\tau]\!]} [\![e]\!]([\![\rho]\!]k) && \text{(by Lemma 4.5(1))} \\
&\leqslant_{[\![\tau]\!]} [\![e']\!]([\![\rho]\!]k) && \text{(by IH)} \\
&\leqslant_{[\![\tau]\!]} [\![e'[\rho]]\!]k && \text{(by Lemma 4.5(2))}
\end{aligned}$$

The code for this case is as follows:

```
sound {Γ} {τ} ._ ._ (ren-cong {.Γ} {Γ'} {.τ} {e} {e'} {ρ} d) k =
  Preorder-str.trans (snd [τ] t)
    (Monotone.f (interpE (ren e ρ)) k)
    (Monotone.f (interpE e) (Monotone.f (interpR ρ) k))
```

```
(Monotone.f (interpE (ren e' ρ)) k)
  (ren-eq-l ρ e k)
  (Preorder-str.trans (snd [τ] t)
    (Monotone.f (interpE e) (Monotone.f (interpR ρ) k))
    (Monotone.f (interpE e') (Monotone.f (interpR ρ) k))
    (Monotone.f (interpE (ren e' ρ)) k)
      (sound e e' d (Monotone.f (interpR ρ) k))
      (ren-eq-r ρ e' k))
```

Now consider a judgement for the composition of substitutions:

$$\frac{}{e[\theta_1 \circ \theta_2] \leqslant \mathsf{s}\ (e[\theta_2])[\theta_1]}$$

where θ_1, θ_2 are substitutions, and $e : \tau$. We would like to show $[\![e[\theta_1 \circ \theta_2]]\!]k \leqslant_{[\![\tau]\!]} [\![(e[\theta_2])[\theta_1]k$. At first glance, it may appropriate to try the proof by inducting on the structure of e, but we can avoid this by using commutative properties of substitution composition which we saw above:

$$\begin{aligned}
[\![e[\theta_1 \circ \theta_2]]\!]k &\leqslant_{[\![\tau]\!]} [\![e]\!]([\![\theta_1 \circ \theta_2]\!]k) && \text{(by Lemma 4.5(1))} \\
&\leqslant_{[\![\tau]\!]} [\![e]\!]([\![\theta_2]\!]([\![\theta_1]\!]k)) && \text{(by monotonicity of } [\![e]\!] \text{ and Lemma 4.9(2))} \\
&\leqslant_{[\![\tau]\!]} [\![e[\theta_2]]\!]([\![\theta_1]\!]k) && \text{(by Lemma 4.5(2))} \\
&\leqslant_{[\![\tau]\!]} [\![(e[\theta_2])[\theta_1]]\!]k && \text{(by Lemma 4.5(2))}
\end{aligned}$$

The code for this case in Agda is as follows:

```
sound {Γ} {τ} ._ (subst (subst e Θ2) Θ1) (subst-ss-l Θ1 Θ2 e) k =
  Preorder-str.trans (snd [τ] t)
    (Monotone.f (interpE (subst e (Θ1 ss Θ2))) k)
    (Monotone.f (interpE e) (Monotone.f (interpS (Θ1 ss Θ2)) k))
    (Monotone.f (interpE (subst (subst e Θ2) Θ1)) k)
    (subst-eq-l (Θ1 ss Θ2) e k)
    (Preorder-str.trans (snd [τ] t)
      (Monotone.f (interpE e) (Monotone.f (interpS (Θ1 ss Θ2)) k))
```

```
(Monotone.f (interpE (subst e Θ2)) (Monotone.f (interpS Θ1) k))
(Monotone.f (interpE (subst (subst e Θ2) Θ1)) k)
(Preorder-str.trans (snd [τ] t)
  (Monotone.f (interpE e) (Monotone.f (interpS (Θ1 ss Θ2)) k))
  (Monotone.f (interpE e) (Monotone.f (interpS Θ2) (Monotone.f (interpS Θ1) k)))
  (Monotone.f (interpE (subst e Θ2)) (Monotone.f (interpS Θ1) k))
  (Monotone.is-monotone (interpE e)
    (Monotone.f (interpS (Θ1 ss Θ2)) k)
    (Monotone.f (interpS Θ2) (Monotone.f (interpS Θ1) k))
      (interp-ss-l Θ1 Θ2 k))
  (subst-eq-r Θ2 e (Monotone.f (interpS Θ1) k)))
(subst-eq-r Θ1 (subst e Θ2) k))
```

□

CHAPTER 5

Examples

Once our system is complete, we will not need to refer to the denotational semantics of the complexity language when analyzing source programs. However, since we currently do not have any support for simplifying terms in the complexity language, running source programs all the way to their interpretation in the denotational semantics is useful in that it is fully automated, and gives us insight into how Agda presents recurrences. In `Samples.agda`, we implement many source programs which can be run through to the denotational semantics, including fast (linear) Fibonacci, and insertion sort for lists. In this chapter, we run two functions through to the denotational semantics to give the reader some insight into how Agda can help us automate the process of recurrence extraction.

1. dbl

As a first example, consider a function dbl which doubles a natural number:

```
dbl : Nat → Nat
dbl Z = Z
dbl (S n) = S (S (dbl n))
```

We can encode this in the source language as follows:

```
dbl : ∀ {Γ} → Γ Source.|- (nat ->s nat)
dbl = lam (rec (var i0) z (suc (suc (force (var (iS i0))))))
```

Composing the interpretation and translation functions yields a monotone function

```
monotone
(λ x → 0,
  monotone
```

```
  (λ p₁ → (natrec (1,0) (λ n x₂ → S (fst x₂), S (S (snd x₂))) p₁)),
  (λ a b c → ERASED))
(λ x y z₁ → ERASED)
```

where p_1 is the natural number argument to dbl, x_2 is a pair whose first component is the cost of the recursive call and whose second component is the result of the recursive call. The components of each pair of the natrec expression represent their respective cost-potential components in the complexity language.

Recall that a monotone function in Agda is equipped with a proof that the function is monotone. When running functions through to the semantics, Agda spends most of its time generating the proofs of monotonicity. These can be at least several hundred lines long, so for readability and efficiency, we can redact these proofs so that they just appear ERASED.

2. rev

Consider the function presented in Chapter 3, rev, which reverses a list, with the help of an auxiliary function append:

```
rev : {A : Set} (l : List A) → List A
rev [] = []
rev (x :: xs) = append (rev xs) (x :: [])
  where
    append : {A : Set} (l1 l2 : List A) → List A
    append [] ys = ys
    append (x :: xs) ys = x :: append xs ys
```

In the source language, we encode these as follows:

```
append : ∀ {Γ τ} → Γ Source.|- (list τ ->s (list τ ->s list τ))
append = lam (lam (listrec (var (iS i0)) (var i0) (var i0 ::s force (var (iS (iS i0))))))
rev : ∀ {Γ τ} → Γ Source.|- (list τ ->s list τ)
rev = lam (listrec (var i0) nil (app (app append (force (var (iS (iS i0))))) (var i0 ::s nil)))
```

The result of running rev through the interpretation is as follows:

```
monotone
(λ x → 0,
  monotone
  (λ p₁ →
    fst (lrec p₁ (1, [])
      (λ x₁ x₂ x₃ → S (S (S (fst x₃ +
      fst (lrec (snd x₃) (1, x₁ :: []) (λ x₄ x₅ x₆ → S (fst x₆), x₄ :: snd x₆))))),
      snd (lrec (snd x₃) (1, x₁ :: []) (λ x₄ x₅ x₆ → S (fst x₆), x₄ :: snd x₆))))
  (λ a b c → ERASED)))
(λ x y z₁ → ERASED)
```

which can be compared directly to the recurrence which we deduced by hand in Chapter 3 for rev:

$$
\begin{aligned}
&(0, \lambda l.(\mathsf{rec}(l, \mathsf{Nil} \mapsto (1, \mathsf{Nil}) \\
&\mid \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.(3 + r_c + ((||\ \mathsf{append}\ ||_p\ \ r_p)_p\ \mathsf{Cons}(x, \mathsf{Nil}))_c, \\
&(||\ \mathsf{append}\ ||_p\ \ r_p)_p\ \mathsf{Cons}(x, \mathsf{Nil}))
\end{aligned}
$$

At the moment, running programs through to their interpretation in the semantics is the best way to see the recurrences, as there is currently no rewriting mechanism in the complexity language to simplify terms into a mildly-readable form as shown above. However, in the future, we can try and use Agda's experimental REWRITE pragma to help us achieve some sort of readability in the complexity language.

CHAPTER 6

Related Work

1. Recurrence Relations

Wegbreit's Metric system, described in his seminal contribution [Weg75], is still one of the most advanced automated complexity analysis systems to date. Much of the literature in the field of automated complexity analysis, including [Ros89], [Ben04], and [LM88], and ours, follows his method of extracting recurrences from source programs and then solving them into closed forms to obtain a big-O bound. In addition to providing closed forms which bound the running time of first-order Lisp programs, Metric can take into account probability distributions on the input values for the program and produce closed forms which incorporate this information. However, Wegbreit notes that Metric "can analyze only fairly simple Lisp programs" [Weg75]. Our system does not incorporate probability distributions when producing an output, but once complete, our system will be able to tackle many more programs than Metric.

One of the most popular developments in recent years is Resource Aware ML (RAML) [Hof11]. RAML uses a source language that is similar to ours, but focuses on amortized analysis, with an approach inspired by Hofmann and Jost [HJ03]. This methodology is different from ours: Hoffmann annotates programs with resource variables, and attempts to infer a type based on these annotations. The constraints collected by type inference are solved by a linear programming solver, which provides a typing for the program. This in turn represents an upper bound on the desired resource. A prototype implementation of RAML can be found at `http://raml.tcs.ifi.lmu.de/prototype`, and is pre-loaded with many example programs for user experimentation. However, RAML only allows for first-order functions; Hoffmann simply calls for the use of defunctionalization [Rey72] to convert higher-order functions into an equivalent first-order form which their system can understand. [HS15] extends this work to handle analysis of parallel programs.

In a recent work, Avanzini, Dal Lago, and Moser [ALM15] attempt to use first-order term rewriting systems to analyze higher-order functional programs. Their process involves using program transformations such as defunctionalization, dead code elimination, and uncurrying to turn higher-order programs into a form which first-order provers can recognize. While their approach is unique, it seems that their system can only analyze a limited range of complexities; the website[1] for their work doesn't list any logarithmic-time algorithms. Avanzini et al. also acknowledge that their system doesn't work on some examples due to technological limitations of current first-order provers. A similar recent work by Gimenez and Moser [GM16] uses interaction nets as a computational model for complexity analysis. [GM16] discusses both sequential and parallel analysis of programs, but the approach requires programs be represented as interaction nets.

The ideas in this work stem from Danner, Licata, and Ramyaa [DLR15]. This thesis formalizes much of the work given in [DLR15], and extends it with a proof of soundness for the denotational semantics we give to the complexity language.

2. Proof Assistants

Danner, Paykin, and Royer [DPR13] approached our problem using Coq [Coq09]. Our approach to the proof of the bounding theorem, presented in [Hud15], is inspired by their work. However, the main difference is that the bounding theorem we prove is *syntactic*, as opposed to the semantic version in [DPR13]. In addition, Agda's support for inductive-recursive definitions [Dyb00] simplifies the central definition of bounding.

Charguéraud and Pottier [CP15] formalize a union-find data structure in Coq to verify its time complexity properties. Although this is not work towards a tool to automate complexity analysis, it is one of the first uses of proof assistants to verify complexity properties of algorithms . Charguéraud and Pottier use time credits and separation logic to achieve their result.

CerCo [AAB+14] is an open-source project which has developed a technique for low-level static program analysis verified by Matita [ARCT11], a proof assistant developed at the University of Bologna. The system's main component is a C compiler, verified in Matita, that

[1] `http://cbr.uibk.ac.at/tools/hoca/`

provides cost annotations for code written in C, which are then fed through an external plugin which in turn synthesizes assertions about the complexity of the program. These assertions are then reduced by the CVC3 automatic theorem prover [BT07] to provide a time bound for the program.

Danielsson [Dan08] presents a simple library for complexity analysis in Agda, with an emphasis on amortized cost. However, their approach requires the user to annotate programs explicitly with cost information, unlike our approach which does so automatically.

CHAPTER 7

Conclusion

We have presented a system for computer-checked recurrence extraction for functional programs, in support of the thesis statement presented at the start of this thesis:

> *It is possible to extract and formally reason about time complexity properties of functional programs using proof assistants.*

In Agda, we extract recurrences from source programs using a monadic translation to a complexity language, and prove that the costs extracted are upper bounds on the cost to evaluate the original program. To validate the complexity language, we give a mathematical model which interprets complexity types as preorders, and terms as monotone functions between preorders. We prove that the complexity language is sound with respect to its interpretation, and exploit the Agda encoding of the interpretation to confirm that the recurrences extracted using our method indeed resemble recurrences extracted using traditional methods.

With the end goal of the project in mind, the work presented in this thesis provides the basis for an automated complexity analysis system embedded within Agda. However, to obtain asymptotic bounds on the running time of source programs, it is necessary to be able to reduce recurrences into closed forms; this is an immediate direction for future work. Two ideas we can explore in order to do this include:

(1) Adding rules to the abstract preorder judgement of the complexity language which will allow us to massage recurrences into closed forms. This implies adding cases to the soundness proof of Chapter 4 to ensure that each of the rules introduced is sound with respect to its semantic interpretation.

(2) Experimenting with Agda's new REWRITE pragma to reduce terms in the complexity language into a legible format, and aid in the process of massaging recurrences into closed forms.

While Agda undoubtedly provides a solid framework on which to construct our system, we must eventually also consider its practicality and ease of use in real life. This is not an immediate goal, but it is agreeable that it would be easier for users to be able to implement functions in a more familiar ML-style syntax, instead of the current source language syntax. For example, implementing dbl as

```
let rec dbl (n : Nat) : Nat =
  match n with
  | 0 → 0
  | S n' → S S (dbl n')
```

instead of

```
dbl : ∀ {Γ} → Γ Source.⊢ (nat ->s nat)
dbl = lam (rec (var i0) z (suc (suc (force (var (iS i0))))))
```

This would at the very least require a parser for user-submitted expressions.

The results we observe from the work presented in this thesis, together with the various components which require future work, suggest that a computer-checked system for automated complexity analysis is indeed a feasible long-term goal, and gives valuable insight into the capabilities of proof assistants as a vehicle for programming language research.

Bibliography

[AAB+14] RobertoM. Amadio, Nicolas Ayache, Francois Bobot, JaapP. Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, DominicP. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. Certified complexity (cerco). In Ugo Dal Lago and Ricardo Peña, editors, *Foundational and Practical Aspects of Resource Analysis*, volume 8552 of *Lecture Notes in Computer Science*, pages 1–18. Springer International Publishing, 2014. URL: http://dx.doi.org/10.1007/978-3-319-12466-7_1, doi:10.1007/978-3-319-12466-7_1.

[ALM15] Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the complexity of functional programs: Higher-order meets first-order (long version). *CoRR*, abs/1506.05043, 2015. URL: http://arxiv.org/abs/1506.05043.

[ARCT11] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The matita interactive theorem prover. In *Automated Deduction–CADE-23*, pages 64–69. Springer, 2011.

[Ben04] Ralph Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004. URL: http://dx.doi.org/10.1016/j.tcs.2003.10.022, doi:10.1016/j.tcs.2003.10.022.

[BG96] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 213–225, New York, NY, USA, 1996. ACM. URL: http://doi.acm.org/10.1145/232627.232650, doi:10.1145/232627.232650.

[BHKM12] Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor Mcbride. Strongly typed term representations in coq. *J. Autom. Reason.*, 49(2):141–159, August 2012. URL: http://dx.doi.org/10.1007/s10817-011-9219-0, doi:10.1007/s10817-011-9219-0.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany, http://www.cs.nyu.edu/acsys/cvc3/.

[Coq09] The Coq Development Team. The coq proof assistant reference manual, 2009. URL: https://coq.inria.fr/.

[CP15] Arthur Charguéraud and François Pottier. Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. *to appear in International Conference on Interactive Theorem Proving (ITP) 2015*, August 2015.

[Dan08] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 133–144, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1328438.1328457, doi:10.1145/1328438.1328457.

[dB72] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972. URL: http://www.sciencedirect.com/science/article/pii/1385725872900340, doi:http://dx.doi.org/10.1016/1385-7258(72)90034-0.

[DLR15] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 140–151, New York, NY, USA, 2015. ACM. URL: http://doi.

acm.org/10.1145/2784731.2784749, doi:10.1145/2784731.2784749.

[DPR13] Norman Danner, Jennifer Paykin, and James S. Royer. A static cost analysis for a higher-order language. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, pages 25–34, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2428116.2428123, doi:10.1145/2428116.2428123.

[Dyb00] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, 65:525–549, 6 2000. URL: http://journals.cambridge.org/article_S0022481200012044, doi:10.2307/2586554.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 163–179, 2013. URL: http://dx.doi.org/10.1007/978-3-642-39634-2_14, doi:10.1007/978-3-642-39634-2_14.

[Geu09] Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[GM16] Stéphane Gimenez and Georg Moser. The complexity of interaction. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 243–255, 2016. URL: http://doi.acm.org/10.1145/2837614.2837646, doi:10.1145/2837614.2837646.

[Gon07] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, page 333, 2007. URL: http://dx.doi.org/10.1007/978-3-540-87827-8_28, doi:10.1007/978-3-540-87827-8_28.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 185–197, New York, NY, USA, 2003. ACM. URL: http://doi.acm.org/10.1145/604131.604148, doi:10.1145/604131.604148.

[Hof11] Jan Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, Ludwig-Maximilians-Universiät München, 2011.

[HS15] Jan Hoffmann and Zhong Shao. *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, chapter Automatic Static Cost Analysis for Parallel Programs, pages 132–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. URL: http://dx.doi.org/10.1007/978-3-662-46669-8_6, doi:10.1007/978-3-662-46669-8_6.

[Hud15] Bowornmet Hudson. Certified Cost Bounds in Agda: A Step Towards Automated Complexity Analysis. Undergraduate honors thesis, Wesleyan University, 2015.

[LM88] Daniel Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, April 1988. URL: http://doi.acm.org/10.1145/42190.42347, doi:10.1145/42190.42347.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991. URL: http://dx.doi.org/10.1016/0890-5401(91)90052-4, doi:10.1016/0890-5401(91)90052-4.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[Plo73] Gordon D Plotkin. *Lambda-definability and logical relations*. School of Artificial Intelligence, University of Edinburgh, 1973.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72,

pages 717–740, New York, NY, USA, 1972. ACM. URL: http://doi.acm.org/10.1145/800194.805852, doi:10.1145/800194.805852.

[Ros89] Mads Rosendahl. Automatic complexity analysis. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 144–156, New York, NY, USA, 1989. ACM. URL: http://doi.acm.org/10.1145/99370.99381, doi:10.1145/99370.99381.

[Sta85] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65(2):85–97, 1985.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[Weg75] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, September 1975. URL: http://doi.acm.org/10.1145/361002.361016, doi:10.1145/361002.361016.