University of Oxford

Third Year Project

Defunctionalization of Higher-Order Constrained Horn Clauses

Long Thanh Pham

Keble College

supervised by Prof. Luke Ong and Dr. Steven Ramsay

A thesis submitted for the degree of

Bachelor of Arts Computer Science

Trinity 2018

Abstract

Building on the successes of satisfiability modulo theories (SMT), Bjørner et al. [2012] initiated a research programme advocating Horn constraints as a suitable basis for automatic program verification. The notion of first-order constrained Horn clauses has recently been extended to higher-order logic by Cathcart Burn et al. [2018]. To exploit the remarkable efficiency of SMT solving, a natural approach to solve systems of higher-order Horn constraints is to reduce them to systems of first-order Horn constraints. This report presents a defunctionalization algorithm to achieve the reduction.

Given a well-sorted higher-order constrained Horn clause (HoCHC) problem instance, the defunctionalization algorithm constructs a first-order well-sorted constrained Horn clause problem. In addition to well-sortedness of the algorithm's output, we prove that if an input HoCHC is solvable, then the result of its defunctionalization is solvable. The converse also holds, which we prove using a recent result on the continuous semantics of HoCHC. To our knowledge, this defunctionalization algorithm is the first sound and complete reduction from systems of higher-order Horn constraints to systems of first-order Horn constraints.

We have constructed DefMono[1], a prototype implementation of the defunctionalization algorithm. It first defunctionalizes an input HoCHC problem and then feeds the result into a backend SMT solver. We have evaluated the performance of DefMono empirically by comparison with two other higher-order verification tools.

[1]The web interface is available at http://mjolnir.cs.ox.ac.uk/dfhochc/.

Acknowledgements

First and foremost, I would like to express my sincerest gratitude to Prof. Luke Ong and Dr. Steven Ramsay for their guidance.

I feel very fortunate to have Prof. Luke Ong as my mentor, who gave me an opportunity to work on a research project during the summer vacation before the third-year. This summer research project proved to be immensely helpful to me, allowing me to acquire all the necessary knowledge and develop the foundational ideas for my third-year project. Furthermore, I very much appreciate the regular meetings with Prof. Ong and his quick and critical feedback to my progress reports.

I am also deeply indebted to Dr. Steven Ramsay. When I was in the stage of learning, he spared his precious time to answer my questions and always showed willingness to help me whenever needed. For all of my progress reports, he scrutinised them, giving me thorough and incisive feedback. I am always amazed by his good eye for detail, his speed of understanding, and his sheer intellectual acuity.

I would like to extend my gratitude to three of Prof. Ong's DPhil students: Toby Cathcart Burn, Jerome Jochems, and Carol Mak. Toby helped me a lot with the implementation, letting me reuse some of his source code and setting up an account on a web server for me. Jerome shared with me his work on the continuous semantics of HoCHC. Carol helped me with creating a cover page in LaTeX.

Contents

Chapter 1

Introduction

1.1 Background

Notwithstanding the existence of undecidable problems, over the past decades, formal verification has proved to be useful and even essential to a number of computing applications. Hardware industries, particularly the semiconductor industry, have long embraced the verification technology because the cost of manufacturing faulty hardware products is too costly. Hence, in such industries, formal verification has been used to detect bugs in early development stages. By contrast, software formal verification had been less widely used than hardware verification because more advanced verification technology is required due to increased complexity in software. However, recent advances in the theory and practice of formal verification have led to wider use of formal methods in software as well. Recognising the value of formal verification, the 2007 Turing Awards were given to Edmund Melson Clarke, E. Allen Emerson, and Joseph Sifakis for their contributions to model checking.

Amongst the enabling technologies in the development of formal verification is satisfiability modulo theories (SMT) solvers [Beckert and Hähnle, 2014]. Many approaches in formal verification reduce input programs to first-order constraints such as loop invariants and dependent types [Bjørner et al., 2012]. These constraints are then fed into SMT solvers to check their satisfiability with respect to certain background theories. The standardisation of input formats for SMT solvers is instrumental in accelerating the development of SMT solvers, allowing larger collections of benchmarks to be built. Also, with respect to formal verification, the standardisation of SMT problem formats achieves separation of concerns by dividing the verification process into constraint generation and SMT solving.

Motivated by the standardisation of SMT problem formats, Bjørner et al. [2012] propose standardization at a higher level: first-order verification problems. They suggest the use of constrained Horn clauses to express first-order verification problems, and their claim that Horn clauses serve as a suitable format of first-order verification problems is substantiated in [Bjørner et al., 2015]. First-order constrained Horn clauses are subsequently extended to higher-order logic by Cathcart Burn et al. [2018].

Whilst numerous verification techniques and tools have been created to verify first-order constrained Horn clauses, higher-order constrained Horn clause problems have not seen as much progress as first-order ones. We can exploit the advances in first-order Horn-clause solving by reducing higher-order constrained Horn clause problems to semantically equivalent first-order ones. This approach is pursued by Cathcart Burn et al. [2018] using refinement types. In this refinement type-based approach, each free top-level relational variable is associated with a type. A valid type assignment can then be thought of as a model of an input HoCHC problem. One drawback of this method is incompleteness. Cathcart Burn et al. [2018] reports an instance of solvable HoCHC for which the refinement type-based approach produces an untypable logic program (that is, no model is found by this approach).

In this work, I take a different approach and develop a defunctionalization algorithm to reduce higher-order constrained Horn clauses to first order ones. This is inspired by Reynolds's defunctionalization, a well-established method of reducing higher-order functional programs to first-order ones.

1.2 Related work

First-order constrained Horn clause problems Using first-order Horn clauses to express first-order verification problems was originally proposed by Bjørner et al. [2012]. They maintain that the Horn clause can serve as a suitable standard format of verification problems, enabling the development of a larger collection of benchmarks in the same format. In [Bjørner et al., 2015], they explain the relationship between Horn clauses and existential fixed-point logic (E+LFP), which is equivalent to Hoare logic. They also provide an overview of how to obtain first-order Horn clauses from first-order programs and how to solve first-order Horn clauses. The paper also gives a number of pointers to more detailed accounts of various Horn-clause verification methods.

Higher-order Horn clause problems and refinement types Cathcart Burn et al. [2018] have extended the notion of constrained Horn clause problems to higher-order logic, and introduced the monotone semantics. Unlike the standard semantics, Horn clause problems have canonical models in monotone semantics, which is a very useful property in automated formal verification. As an alternative representation of the higher-order constrained Horn clause problem, the monotone safety problem is introduced. Unlike the Horn clause problem, the monotone safety problem does not contain logical implication, which is not monotone. Thus, the monotone safety problem is a more suitable representation in the monotone semantics, although the difference between the monotone safety problem and Horn clause problem is purely syntactic. The paper also explores the connection between the standard and monotone semantics, proving that any higher-order constrained Horn clause problem in the standard semantics can be converted into a semantically equivalent monotone safety problem.

In the second half of the paper, a refinement type-based approach to verifying monotone safety problems is presented.

Defunctionalization In the conclusion of [Cathcart Burn et al., 2018], Burn et al. propose the use of Reynolds's defunctionalization to reduce higher-order Horn clause problems to first-order ones as done by the refinement type-based approach. This is what motivates the present work on defunctionalization of HoCHC. The idea of representing higher-order functions by closures to verify higher-order programs can also be found in [Bjørner et al., 2013], although this only gives a brief overview of the approach.

Defunctionalization is explained in a detailed yet readable manner in its original paper by Reynolds [Reynolds, 1972]. In this paper, typability of the apply function created as a result of defunctionalization is not considered. A problem arises when we deal with polymorphic languages. This issue is resolved using type specialization in [Bell et al., 1997]. Another work on defunctionalization of polymorphic languages is [Pottier and Gauthier, 2004]. Although the present work on defunctionalization of monotone safety problems does not involve polymorphic types, the idea of formulating a defunctionalization algorithm using inference rules comes from [Pottier and Gauthier, 2004].

1.3 Contributions

The chief contribution of this work is the development of a defunctionalization algorithm to reduce HoCHC to first-order constrained Horn clauses. With respect to the correctness of the algorithm, I prove type preservation, completeness, and soundness. The output of the defunctionalization algorithm is proved to be well-sorted, given that the input is well-sorted. Using the idea of valuation extraction, I also prove that if an input higher-order constrained Horn clause problem is solvable, then its defunctionalized problem is also solvable. The proof for the converse is achieved by using a recent result on the continuous semantics of HoCHC [Jochems, 2018]. As far as I am aware, this is the first sound and complete reduction from HoCHC to first-order constrained Horn clauses.

1.4 Outline of this report

This document is structured as follows.

Chapter 2 introduces higher-order logic, logic program safety problems, and monotone semantics.

Chapter 3 illustrates how defunctionalization works on a concrete example.

In the first half of Chapter 4, the defunctionalization algorithm is formulated using inference rules. In the second half of this chapter, completeness and soundness of the algorithm are established.

Chapter 5 presents a prototype tool based on the defunctionalization algorithm and compares its performance with other higher-order verification tools.

Chapter 6 summarises the work and proposes a few directions for future work.

Appendix A presents details of the preprocessing in the defunctionalization algorithm. Also, the rationale for the algorithm's design is given.

Appendix B describes how to obtain monotone valuations for outputs of the defunctionalization algorithm.

Appendix C provides detailed proofs for the lemmas and theorems presented in Chapter 4.

Appendix D gives a formal proof of type preservation.

The word count of the body of this report is 9995.

Chapter 2

Preliminaries

This chapter introduces the basics of higher-order logic, logic program safety problems, and monotone semantics. Higher-order constrained Horn clauses (HoCHC) are not formally introduced, since the defunctionalization algorithm works on logic program safety problems, which are alternative representations of HoCHC [Cathcart Burn et al., 2018]. It is therefore sufficient to understand that HoCHC and logic program safety problems are equivalent.

2.1 Higher-order logic

In this section I review the syntax and semantics of higher-order logic based on a simply typed lambda calculus. The presentation style of this section follows the one in [Cathcart Burn et al., 2018].

2.1.1 Syntax

In a simply typed lambda calculus, each value is associated with a sort that denotes the category of elements to which the value belongs. Let $(b \in)\mathbb{B}$ be a fixed set of user-defined base sorts including a sort ι of individuals and a sort o of propositions. Using the base sorts, simple sorts are inductively defined as follows:

$$\sigma ::= b \mid \sigma_1 \to \sigma_2,$$

where $b \in \mathbb{B}$. As standard, the sort constructor $\to$ associates to the right. The order of a sort is defined by

$$\begin{aligned} \mathtt{ord}(b) &= 1 && \text{if } b \in \mathbb{B} \\ \mathtt{ord}(\sigma_1 \to \sigma_2) &= \max\{\mathtt{ord}(\sigma_1) + 1, \mathtt{ord}(\sigma_2)\} && \text{otherwise.} \end{aligned}$$

Let $\Sigma = (\mathbb{B}, \mathbb{S})$ denote a first-order signature, where $\mathbb{B}$ is a set of base sorts that includes the propositional sort o and at least one sort of individuals. $\mathbb{S}$ is a set of constant symbols, each of which is associated with a first-order sort (i.e. a sort whose

order is at most 2). As $\mathbb{S}$ can be viewed as a mapping from constant symbols to simple sorts, I write $\mathbb{S}(c)$ for the sort assigned to c by $\mathbb{S}$. Note that because a lambda calculus does not distinguish between functions and values of base sorts, 'constant symbols' in $\mathbb{S}$ include not only those symbols with base sorts but also symbols of arrows types; i.e. function symbols.

Given $\Sigma = (\mathbb{B}, \mathbb{S})$, terms are inductively defined by

$$M, N ::= x \mid c \mid M\ N \mid \lambda x{:}\sigma.M,$$

where x is a variable and $c \in \mathbb{S}$. Standardly, function application associates to the left. Also, the scopes of lambda abstractions extend as far to the right as possible. If a term M has sort $\sigma_1 \to \cdots \to \sigma_m \to b$, where $b \in \mathbb{B}$, the arity of M is defined as

$$\mathbf{ar}(M) = m.$$

The set of free variables occurring in term M is denoted by $\mathtt{FV}(M)$.

A sort environment Δ is a finite sequence of pairs $x : \sigma$, where x is a variable and σ is a simple type. The sort environment is required to have no conflicts; that is, it must not assign multiple sorts to the same variable. The sorts of terms are defined by the following sorting rules:

$$(\textsc{SCst})\ \frac{}{\Delta \vdash c : \mathbb{S}(c)} \qquad (\textsc{SVar})\ \frac{}{\Delta_1, x : \sigma, \Delta_2 \vdash x : \sigma}$$

$$(\textsc{SApp})\ \frac{\Delta \vdash s : \sigma_1 \to \sigma_2 \qquad \Delta \vdash t : \sigma_1}{\Delta \vdash s\ t : \sigma_2}$$

$$(\textsc{SAbs})\ \frac{\Delta, x : \sigma_1 \vdash s : \sigma_2}{\Delta \vdash \lambda x{:}\sigma_1.s : \sigma_1 \to \sigma_2}\ x \notin \mathrm{dom}(\Delta)$$

Notice that the sorts of constant symbols are specified by a signature, whilst the sorts of free variables are specified by a sort environment.

Next, to define formulas of higher-order logic, logical connectives are introduced as constant symbols outside Σ. Let $\mathtt{LSym}$ be the set of the following logical constant symbols:

$$\mathtt{true}, \mathtt{false} : o \qquad \neg : o \to o$$
$$\wedge, \vee, \Rightarrow : o \to o \to o \qquad \forall_\sigma, \exists_\sigma : (\sigma \to o) \to o.$$

I adopt the convention that $\exists_\sigma(\lambda x{:}\sigma.M)$ is shortened to $\exists x{:}\sigma.M$ or $\exists_\sigma x.M$. Furthermore, if the sort of x is clear from the context, $\exists x.M$ can be written.

Formulas are defined as well-sorted terms that have the sort o and whose constant symbols are from either $\mathbb{S}$ or $\mathtt{LSym}$.

Lastly, relational sorts are formally defined by

$$\rho ::= o \mid b \to o \mid \rho_1 \to \rho_2,$$

where $b \in \mathbb{B}$.

2.1.2 Semantics

Given a first-order signature $\Sigma = (\mathbb{B}, \mathbb{S})$, a structure A assigns a non-empty set of elements A_ι to each $\iota \in \mathbb{B}$, where $\iota \neq o$. The sets A_ι are often called universes. To the sort o is assigned the distinguished lattice $\mathbb{2} = \{0 \leq 1\}$. The full sort frame over A is defined inductively on a sort as follows:

$$\begin{aligned} \mathcal{S}[\![\iota]\!] &:= A_\iota && \iota \in \mathbb{B}, \iota \neq o \\ \mathcal{S}[\![o]\!] &:= \mathbb{2} \\ \mathcal{S}[\![\sigma_1 \to \sigma_2]\!] &:= \mathcal{S}[\![\sigma_1]\!] \Rightarrow \mathcal{S}[\![\sigma_2]\!], \end{aligned}$$

where $X \Rightarrow Y$ is the full set-theoretic function space between sets X and Y. To each constant symbol c in $\mathbb{S}$, A assigns an element from $\mathcal{S}[\![\mathbb{S}(c)]\!]$. Let c^A denote this element.

The lattice $\mathbb{2}$ supports the following functions:

$$\begin{aligned} \texttt{or}(b_1)(b_2) &= \max\{b_1, b_2\} & \texttt{not}(b) &= 1 - b \\ \texttt{and}(b_1)(b_2) &= \min\{b_1, b_2\} & \texttt{implies}(b_1)(b_2) &= \texttt{or}(\texttt{not}(b_1))(b_2) \\ \texttt{exists}_\sigma(f) &= \max\{f(v) \mid v \in \mathcal{S}[\![\sigma]\!]\} & \texttt{forall}_\sigma(f) &= \texttt{not}(\texttt{exists}_\sigma(\texttt{not} \circ f)). \end{aligned}$$

For each logical constant symbol $c \in \texttt{LSym}$, I denote the corresponding Boolean function given above by $c^{\texttt{LFun}}$.

The order on $\mathbb{2}$ can be extended to define an order $\subseteq_\rho$ on $\mathcal{S}[\![\rho]\!]$, where ρ is a relational sort:

- For all $b_1, b_2 \in \mathcal{S}[\![o]\!]$, if $b_1 \leq b_2$, then $b_2 \subseteq_o b_2$;
- For all $r_1, r_2 \in \mathcal{S}[\![b \to \rho]\!]$, if $r_1(n) \subseteq_\rho r_2(n)$ for all $n \in \mathcal{S}[\![b]\!]$, then $r_1 \subseteq_{b \to \rho} r_2$;
- For all $r_1, r_2 \in \mathcal{S}[\![\rho_1 \to \rho_2]\!]$, if $r_1(s) \subseteq_\rho r_2(s)$ for all $s \in \mathcal{S}[\![\rho_1]\!]$, then $r_1 \subseteq_{\rho_1 \to \rho_2} r_2$.

The full sort frame can be defined on a sort environment Δ using an indexed Cartesian product:

$$\mathcal{S}[\![\Delta]\!] := \prod_{x \in \mathrm{dom}(\Delta)} \mathcal{S}[\![\Delta(x)]\!].$$

In other words, this is the set of all functions mapping each variable x in $\mathrm{dom}(\Delta)$ to an element in $\mathcal{S}[\![\Delta(x)]\!]$. These functions are called valuations. The order on $\mathcal{S}[\![\Delta]\!]$ can be defined in the same fashion as above: for all $f_1, f_2 \in \mathcal{S}[\![\Delta]\!]$, if $f_1(x) \subseteq_\rho f_2(2)$ for all $x : \rho \in \Delta$, then $f_1 \subseteq_\Delta f_2$.

The interpretation of a term $\Delta \vdash M : \sigma$ is given by an inductively defined function $\mathcal{S}[\![\Delta \vdash M : \sigma]\!] : \mathcal{S}[\![\Delta]\!] \Rightarrow \mathcal{S}[\![\sigma]\!]$. When M consists only of one symbol, $\mathcal{S}[\![\Delta \vdash M : \sigma]\!]$ is defined by

$$\begin{aligned} \mathcal{S}[\![\Delta \vdash x : \sigma]\!](\alpha) &= \alpha(x) && \text{if } x \text{ is a variable} \\ \mathcal{S}[\![\Delta \vdash c : \sigma]\!](\alpha) &= c^A && \text{if } c \in \mathbb{S} \\ \mathcal{S}[\![\Delta \vdash c : \sigma]\!](\alpha) &= c^{\texttt{LFun}} && \text{otherwise,} \end{aligned}$$

where α is a valuation from $\mathcal{S}[\![\Delta]\!]$. If M has a compound structure, we have

$$\mathcal{S}[\![\Delta \vdash M\ N : \sigma_2]\!](\alpha) = \mathcal{S}[\![\Delta \vdash M : \sigma_1 \to \sigma_2]\!](\alpha)(\mathcal{S}[\![\Delta \vdash N : \sigma_1]\!](\alpha))$$
$$\mathcal{S}[\![\Delta \vdash \lambda x{:}\sigma_1.M : \sigma_1 \to \sigma_2]\!](\alpha) = \lambda v \in \mathcal{S}[\![\sigma_1]\!].\mathcal{S}[\![\Delta, x : \sigma_1 \vdash M : \sigma_2]\!](\alpha[x \mapsto v]).$$

Notice that the interpretation of non-logical constant symbols is given by a structure, whereas the interpretation of free variables is given by a valuation.

Assume we are given a Σ-structure A, a formula $\Delta \vdash M : o$, and a valuation $\alpha \in \mathcal{S}[\![\Delta]\!]$. Then $\langle A, \alpha \rangle$ satisfies M if and only if $\mathcal{S}[\![\Delta \vdash M : o]\!](\alpha) = 1$. This satisfaction relation is denoted by $A, \alpha \vDash M$.

2.2 Logic program safety problems

Each verification problem comprises two components: a definite formula component, which describes an input program, and a goal formula component, which is the property of the input program that we want to verify. This section introduces verification problems whose definite formula components are expressed using logic programs. Again, the presentation style of this section follows that in [Cathcart Burn et al., 2018].

2.2.1 Constraint languages

Given a first-order signature Σ, a constraint language is defined as (Tm, Fm, Th), where Tm is a distinguished subset of first-order terms that can be built from Σ, Fm is a distinguished subset of first-order formulas that can be built from Σ, and Th is a theory in which to interpret Fm. Any formula from Fm is called a constraint and Th is called a background theory. We allow Tm and Fm to be strict subsets of all terms and formulas built from Σ as some background theories only consider strict subsets of formulas; e.g. quantifier-free formulas.

In this document, formulas in a constraint language refer to terms of sort o. Therefore, we have $Fm \subseteq Tm$, unlike in usual presentations of predicate logic, where $Tm \cap Fm = \emptyset$.

2.2.2 Goal terms

Fix a first-order signature $\Sigma = (\mathbb{B}, \mathbb{S})$ and a constraint language (Tm, Fm, Th) over Σ. The class of well-sorted goal terms $\Delta \vdash G : \rho$, where ρ is a relational sort, is given by these sorting rules:

$$(\textsc{GCst})\ \frac{}{\Delta \vdash c : \rho_c}\ c \in \{\wedge, \vee, \exists_\iota\} \cup \{\exists_\rho \mid \rho\} \qquad (\textsc{GVar})\ \frac{}{\Delta_1, x : \rho, \Delta_2 \vdash x : \rho}$$

$$(\textsc{GConstr})\ \frac{}{\Delta \vdash \varphi : o}\ \Delta \vdash \varphi : o \in Fm$$

$$(\textsc{GAbs})\ \frac{\Delta, x : \sigma \vdash G : \rho}{\Delta \vdash \lambda x{:}\sigma.G : \sigma \to \rho}\ x \notin \mathtt{dom}(\Delta)$$

$$(\textsc{GAppl})\ \frac{\Delta \vdash G : b \to \rho}{\Delta \vdash G\ N : \rho}\ \Delta \vdash N : b \in Tm$$

$$(\textsc{GAppR})\ \frac{\Delta \vdash G : \rho_1 \to \rho_2 \qquad \Delta \vdash H : \rho_1}{\Delta \vdash G\ H : \rho_2}$$

Throughout the above six rules, b denotes a base sort from $\mathbb{B}$, ρ (with or without subscripts) denotes a relational sort, and σ is either a base sort or a relational sort. Henceforth, I assume that goal terms are well-sorted.

2.2.3 Logic programs

Assume that a first-order signature and a constraint language are fixed. A higher-order constrained logic program P over a sort environment $\Delta = x_1 : \rho_1, \ldots, x_m : \rho_m$, where each ρ_i is a relational sort, is a finite system of (mutual) recursive definitions of shape:

$$x_1{:}\rho_1 = G_1, \ldots, x_m{:}\rho_m = G_m,$$

where each G_i is a goal term and each x_i is distinct. I will call each x_i a top-level relational variable. P is said to be well-sorted whenever $\Delta \vdash G_i : \rho_i$ (i.e. G_i is well-sorted and has relational sort ρ_i) for each $1 \leq i \leq m$. It follows that if P is well-sorted,

$$\mathtt{FV}(G_i) \subseteq \{x_1, \ldots, x_m\}$$

for all $1 \leq i \leq m$.

Since each x_i is distinct, we can regard P as a finite map from variables to goal terms. Thus, let $P(x_i)$ denote the goal term G_i that is bound to x_i. I write $\vdash P : \Delta$ to mean that P is a well-sorted program over Δ.

To interpret logic programs, I use the standard semantics. Let A be a Σ-structure and P be a well-sorted logic program over a sort environment Δ. The one-step consequence operator of P is the functional $T^{\mathcal{S}}_{P:\Delta} : \mathcal{S}[\![\Delta]\!] \Rightarrow \mathcal{S}[\![\Delta]\!]$ defined by

$$T^{\mathcal{S}}_{P:\Delta}(\alpha)(x) = \mathcal{S}[\![\Delta \vdash P(x) : \Delta(x)]\!](\alpha).$$

A valuation α is a prefixed point of $T^{\mathcal{S}}_{P:\Delta}$ if and only if we have $T^{\mathcal{S}}_{P:\Delta}(\alpha) \subseteq_\Delta \alpha$.

2.2.4 Logic program safety problems

Suppose that Σ is a first-order signature and $L = (Tm, Fm, Th)$ is a constraint language over Σ. A logic program safety problem is defined as a triple (Δ, P, G), where Δ is a sort environment of relational variables, P is a well-sorted logic program over Δ, and G is a goal term that has sort o and is built from Σ and L. The problem is solvable if and only if for all models of Th, there exists a valuation α such that α is a prefixed point of $T^{\mathcal{S}}_{P:\Delta}$ and $\mathcal{S}[\![\Delta \vdash G : o]\!](\alpha) = 0$. G is usually the negation of a property that we want P to satisfy.

2.3 Monotone semantics

The monotone semantics for logic programs is introduced by Cathcart Burn et al. [2018] as an alternative to the standard semantics. The importance of the monotone semantics in defunctionalization will be explained in Section 3.3.

2.3.1 Semantics

Given a first-order signature $\Sigma = (\mathbb{B}, \mathbb{S})$, structure A assigns a non-empty discrete poset A_ι to each $\iota \in \mathbb{B}$, where $\iota \neq o$. As in the standard semantics, A assigns $\mathbb{2}$ to the sort o. Discrete posets are defined as partially ordered sets in which no two distinct elements are comparable. The monotone sort frame over A is then inductively defined as

$$\mathcal{M}[\![\iota]\!] := A_\iota \qquad \mathcal{M}[\![o]\!] := \mathbb{2} \qquad \mathcal{M}[\![\sigma_1 \to \sigma_2]\!] := \mathcal{M}[\![\sigma_1]\!] \Rightarrow_m \mathcal{M}[\![\sigma_2]\!],$$

where $X \Rightarrow_m Y$ is the monotone function space between posets X and Y. The universe A_ι is regarded as a discrete "poset" rather than simply a set because we want the definition $\mathcal{M}[\![\sigma_1 \to \sigma_2]\!] := \mathcal{M}[\![\sigma_1]\!] \Rightarrow_m \mathcal{M}[\![\sigma_2]\!]$ to encompass the cases when $\sigma_1 \in \mathbb{B}$. Since any set can be considered as a discrete poset, when $\sigma_1 \in \mathbb{B} \setminus \{o\}$, $\Rightarrow_m$ is the same as $\Rightarrow$ in the definition of full sort frames.

A also maps each constant symbol $c : \sigma \in \mathbb{S}$ to an element from $\mathcal{M}[\![\sigma]\!]$.

The order in $\mathbb{2}$ is extended to $\mathcal{M}[\![\rho]\!]$, where ρ is a relational sort, in the same manner as $\mathcal{S}[\![\rho]\!]$. Also, the set of valuations with respect to sort environment Δ is defined analogously to the standard semantics:

$$\mathcal{M}[\![\Delta]\!] := \prod_{x \in \mathrm{dom}(\Delta)} \mathcal{M}[\![\Delta(x)]\!].$$

The monotone interpretation of goal terms is inductively defined in the same way as the standard interpretation. As we consider only monotone functions, the definition of `exists` becomes

$$\texttt{exists}_\sigma(f) = \max\{f(v) \mid v \in \mathcal{M}[\![\sigma]\!]\}.$$

Fix a first-order signature, a constraint language, and a structure for interpretation of a logic program. The one-step consequence operator $T^{\mathcal{M}}_{P:\Delta}$ is defined as

$$T^{\mathcal{M}}_{P:\Delta}(\alpha)(x) = \mathcal{M}[\![\Delta \vdash P(x) : \Delta(x)]\!](\alpha).$$

A prefixed point of $T^{\mathcal{M}}_{P:\Delta}$ is called a model of the logic program P. The term 'model' is overloaded because a model of a logic program is a valuation, whereas a model of a theory is a structure.

2.3.2 Monotone logic program safety problems

Suppose that Σ is a first-order signature and $L = (Tm, Fm, Th)$ is a constraint language over Σ. A monotone logic program safety problem (oftentimes abbreviated as a monotone

problem) is defined as a triple (Δ, P, G), where Δ is a sort environment of relational variables, P is a well-sorted logic program over Δ, and G is a goal formula. Both P and G are built from Σ and L. The monotone problem is solvable if and only if for all models of Th, there exists a valuation α such that α is a prefixed point of $T^{\mathcal{M}}_{P:\Delta}$ and $\mathcal{M}[\![\Delta \vdash G : o]\!](\alpha) = 0$. G is usually the negation of a property that we want P to satisfy.

Theorem 2 in [Cathcart Burn et al., 2018] establishes a bridge between constrained Horn clause problems and monotone logic program safety problems:

Theorem 2.3.1. *A higher-order constrained Horn clause problem (Δ, D, G) is solvable if and only if the associated monotone logic program safety problem (Δ, P_D, G) is solvable.*

The transformation from the definite Horn formula D to the corresponding logic program P_D is provided in Section 4.1 of [Cathcart Burn et al., 2018].

Chapter 3

Defunctionalization of monotone problems

This chapter illustrates how defunctionalization works on a concrete example of a logic program safety problem, which is interpreted using the standard semantics. An issue that arises from higher-order existential quantification is then explained. The monotone semantics is instrumental in resolving this issue.

3.1 Overview

In this section, I will illustrate the workings of the defunctionalization algorithm for logic program safety problems using a concrete example. Because the standard semantics is more natural and intuitive than the monotone semantics, the standard semantics allows us to use our own intuition to interpret safety problems. Consequently, we can follow how defunctionalization proceeds without being concerned about semantics. Therefore, we will use the standard semantics to interpret the example logic program safety problem.

Henceforth, for readability, I omit subscripts of $\exists$ that denote the sorts of quantified variables.

Consider the safety problem $\mathcal{P} = (\Delta, P, G)$, where Δ is given by

$$\begin{aligned}\Delta = \{&Main : \mathbf{nat} \to \mathbf{natlist} \to o,\\ &TwiceMap : (\mathbf{nat} \to \mathbf{nat} \to o) \to \mathbf{natlist} \to \mathbf{natlist} \to o,\\ &Map : (\mathbf{nat} \to \mathbf{nat} \to o) \to \mathbf{natlist} \to \mathbf{natlist} \to o,\\ &Twice : (\mathbf{nat} \to \mathbf{nat} \to o) \to \mathbf{nat} \to \mathbf{nat} \to o\},\end{aligned}$$

P is

$$\begin{aligned}Main &= \lambda n, ns.TwiceMap\ (\lambda a, b.a + n = b)\ (\texttt{cons}\ 0\ \texttt{nil})\ ns\\ TwiceMap &= \lambda f.Map\ (Twice\ f)\\ Map &= \lambda f, a, b.(a = \texttt{nil} \wedge b = \texttt{nil}) \vee\\ &\qquad (\exists n, ns, m, ms.a = \texttt{cons}\ n\ ns \wedge f\ n\ m \wedge Map\ f\ ns\ ms \wedge b = \texttt{cons}\ m\ ms)\\ Twice &= \lambda f, a, b.(\exists c.f\ a\ c \wedge f\ c\ b),\end{aligned}$$

and G is

$$G = \exists n, ns.Main\ n\ ns \wedge ns = \texttt{nil}.$$

The signature for $\mathcal{P}$ is $\Sigma = (\mathbb{B}, \mathbb{S})$, where $\mathbb{B}$ and $\mathbb{S}$ are

$$\begin{aligned}\mathbb{B} &= \{\mathbf{nat}, \mathbf{natlist}, o\}\\ \mathbb{S} &= \{\texttt{nil} : \mathbf{natlist}, \texttt{cons} : \mathbf{nat} \to \mathbf{natlist} \to \mathbf{natlist}, + : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}\\ &\quad =_{\mathbf{natlist}}\!: \mathbf{natlist} \to \mathbf{natlist} \to o, =_{\mathbf{nat}}\!: \mathbf{nat} \to \mathbf{nat} \to o\} \cup \{n : \mathbf{nat} \mid n \in \mathbb{N}\}.\end{aligned}$$

To be precise, $\mathbb{S}$ must be finite. However, having all natural numbers included in $\mathbb{S}$ does not affect the fundamental nature of this verification problem.

Observe that $\{n : \mathbf{nat} \mid n \in \mathbb{N}\}$ is a set of symbols rather than a set of mathematical entities.

As $=_{\mathbf{natlist}}$ and $=_{\mathbf{nat}}$ have different types, they must be distinguished. However, I will denote both of them by $=$ for simplicity. From the sorts of their arguments, we can infer which equality is in use.

The background theory we use to interpret $\mathcal{P}$ is the one constructed by a structure that maps each $c \in \mathbb{S}$ to the naturally corresponding element in the universe of natural numbers and their lists.

Defunctionalization is the conversion of higher-order programs to a semantically equivalent first-order programs. To achieve it, higher-order parameters appearing in the programs need to be removed. Parameters are classified into formal parameters and actual parameters. Higher-order formal parameters in logic programs are always found in lambda abstractions. Higher-order actual parameters are generated by curried functions, which can be (strictly) partially applied. Hence, the higher-order property of logic programs is attributed to

- Higher-order formal parameters
- Curried functions.

Higher-order formal parameters can be identified by looking at type annotations. However, formal parameters can be missing due to currying. For instance, in the logic program P given above, the definition of $TwiceMap$ is a lambda abstraction with only one parameter f. However, since the sort of $TwiceMap$ is $(\mathbf{nat} \to \mathbf{nat} \to o) \to \mathbf{natlist} \to \mathbf{natlist} \to o$, we have $\mathbf{ar}(TwiceMap) = 3$. Therefore, $TwiceMap$ has two more formal parameters. The first preprocessing step is thus to uncover hidden formal parameters. This is known as η-expansion in the literature on lambda calculi. We only uncover the formal parameters of outermost lambda abstractions. In P, the only place where formal parameters of top-level relational variables are hidden is $TwiceMap$.

Applying η-expansion to it, we obtain

$$\begin{aligned}
Main &= \lambda n, ns.TwiceMap\ (\lambda a, b.a + n = b)\ (\texttt{cons}\ 0\ \texttt{nil})\ ns \\
TwiceMap &= \lambda f, xs, ys.Map\ (Twice\ f)\ xs\ ys \\
Map &= \lambda f, a, b.(a = \texttt{nil} \wedge b = \texttt{nil}) \\
&\quad \vee (\exists n, ns, m, ms.a = \texttt{cons}\ n\ ns \wedge f\ n\ m \wedge Map\ f\ ns\ ms \wedge b = \texttt{cons}\ m\ ms) \\
Twice &= \lambda f, a, b.(\exists c.f\ a\ c \wedge f\ c\ b).
\end{aligned}$$

Note that although $Twice\ f$ inside the definition of $TwiceMap$ has a functional sort and hence its formal parameters are hidden, we do not apply η-expansion to it, because it is not an outermost function. Now all formal parameters of outermost lambda abstractions in P are visible.

Curried functions in logic programs appear either as the definitions of top-level relational variables in the form $x_i{:}\rho_i = G_i$ or as anonymous lambda abstractions. The constant symbols from $\mathbb{S}$ cannot be strictly partially applied in a logic program. Hence, we do not need to defunctionalize the functions declared in the signature.

In the logic program above, $\lambda a, b.a + n = b$ inside the definition of $Main$ is an example of anonymous functions. Later in the process of defunctionalization, higher-order actual parameters are replaced with first-order data, each of which is labelled with an associated curried function. Whilst top-level relational variables can be used as labels for the functions that define them, anonymous functions do not have any unique name or variable associated with it. Therefore, for convenience, I create fresh top-level relational variables for anonymous functions so that they are not 'anonymous' anymore. Consequently, P becomes

$$\begin{aligned}
Main &= \lambda n, ns.TwiceMap\ (Add\ n)\ (\texttt{cons}\ 0\ \texttt{nil})\ ns \\
TwiceMap &= \lambda f, ns, ms.Map\ (Twice\ f)\ ns\ ms \\
Map &= \lambda f, a, b.(a = \texttt{nil} \wedge b = \texttt{nil}) \\
&\quad \vee (\exists n, ns, m, ms.a = \texttt{cons}\ n\ ns \wedge f\ n\ m \wedge Map\ f\ ns\ ms \wedge b = \texttt{cons}\ m\ ms) \\
Twice &= \lambda f, a, b.(\exists c.f\ a\ c \wedge f\ c\ b) \\
Add &= \lambda n, a, b.a + n = b.
\end{aligned}$$

Next, higher-order parameters are replaced with first-order data of a new base type. Let us denote the new type **closr**, which is short for 'closure'. Because I will defunctionalize P step by step, the intermediate states may not be valid logic programs.

Partially applied curried functions are represented by algebraic data types. They store all actual parameters that have been supplied so far. It is necessary to define distinct data constructors according to the number of actual parameters. For instance,

partially applied instances of function *Add* can take one of the following forms:

$$\begin{aligned} &\mathit{Add}\\ &\mathit{Add}\ x\\ &\mathit{Add}\ x\ y\\ &\mathit{Add}\ x\ y\ z, \end{aligned}$$

where the last form is not strictly partially applied.

Let C^i_F be a data constructor that, after i many arguments are supplied, represents a partially applied instance of function F with i many actual parameters. To simulate lambda/function application using these constructors, we need to define function *Apply*. For the top-level relational variable $Main$ defined in P, the corresponding *Apply* is given by

$$\begin{aligned} &\mathit{Apply} = \lambda x, y, z.x = C^0_{Main} \wedge z = C^1_{Main}\ y\\ &\mathit{Apply} = \lambda x, ns.(\exists n.x = C^1_{Main}\ n \wedge \mathit{TwiceMap}\ (\mathit{Add}\ n)\ (\texttt{cons}\ 0\ \texttt{nil})\ ns). \end{aligned}$$

Notice that $\mathit{TwiceMap}\ (\mathit{Add}\ n)\ (\texttt{cons}\ 0\ \texttt{nil})\ ns$ in the second line is derived from the definition of $Main$ in P. *Apply* on the first line has sort **closr** $\rightarrow$ **nat** $\rightarrow$ **closr** $\rightarrow o$ and takes three arguments. The first argument represents a partially applied instance of $Main$. The second argument is an input to the function represented by the first argument. The third argument is the result of applying the second argument to the function represented by the first argument.

By contrast, *Apply* on the second line has sort **closr** $\rightarrow$ **natlist** $\rightarrow o$ and has arity 2. The first argument represents a partially applied instance of $Main$ in which only the last parameter of $Main$ is missing. The second argument corresponds to this missing final parameter. In a logic program, if a top-level relational sort F has arity n, the first $n-1$ parameters of F can be interpreted as inputs (as in functional programming) and the last parameter of F can be interpreted as the corresponding output. Using this interpretation, the second *Apply* is considered as linking the input and output of $Main$.

In this way, the first *Apply* function simulates function application, whereas the second *Apply* works out whether the first argument evaluates to the second argument. As these two *Apply* functions have different roles, I rename the second *Apply* to 'IOMatch'. This yields

$$\begin{aligned} \mathit{Apply} &= \lambda x, y, z.x = C^0_{Main} \wedge z = C^1_{Main}\ y\\ \mathit{IOMatch} &= \lambda x, ns.(\exists n.x = C^1_{Main}\ n \wedge \mathit{TwiceMap}\ (\mathit{Add}\ n)\ (\texttt{cons}\ 0\ \texttt{nil})\ ns). \end{aligned}$$

Applying the same step to the remaining top-level relational variables in P gives

$$\begin{aligned}
\mathit{Apply} &= \lambda x, y, z.x = C^0_{\mathit{Main}} \wedge z = C^1_{\mathit{Main}}\ y \\
\mathit{IOMatch} &= \lambda x, ns.(\exists n.x = C^1_{\mathit{Main}}\ n \wedge \mathit{TwiceMap}\ (\mathit{Add}\ n)\ (\texttt{cons}\ 0\ \texttt{nil})\ ns)
\end{aligned}$$

$$\begin{aligned}
\mathit{Apply} &= \lambda x, y, z.x = C^0_{\mathit{TwiceMap}} \wedge z = C^1_{\mathit{TwiceMap}}\ y \\
\mathit{Apply} &= \lambda x, y, z.(\exists f.x = C^1_{\mathit{TwiceMap}}\ f \wedge z = C^2_{\mathit{TwiceMap}}\ f\ y) \\
\mathit{IOMatch} &= \lambda x, ms.(\exists f, ns.x = C^2_{\mathit{TwiceMap}}\ f\ ns \wedge \mathit{Map}\ (\mathit{Twice}\ f)\ ns\ ms)
\end{aligned}$$

$$\begin{aligned}
\mathit{Apply} &= \lambda x, y, z.x = C^0_{\mathit{Map}} \wedge z = C^1_{\mathit{Map}}\ y \\
\mathit{Apply} &= \lambda x, y, z.(\exists f.x = C^1_{\mathit{Map}}\ f \wedge z = C^2_{\mathit{Map}}\ f\ y) \\
\mathit{IOMatch} &= \lambda x, b.(\exists f, a.x = C^2_{\mathit{Map}}\ f\ a \wedge ((a = \texttt{nil} \wedge b = \texttt{nil}) \\
&\qquad \vee (\exists n, ns, m, ms.a = \texttt{cons}\ n\ ns \wedge f\ n\ m \wedge \mathit{Map}\ f\ ns\ ms \wedge b = \texttt{cons}\ m\ ms)))
\end{aligned}$$

$$\begin{aligned}
\mathit{Apply} &= \lambda x, y, z.x = C^0_{\mathit{Twice}} \wedge z = C^1_{\mathit{Twice}}\ y \\
\mathit{Apply} &= \lambda x, y, z.(\exists f.x = C^1_{\mathit{Twice}}\ f \wedge z = C^2_{\mathit{Twice}}\ f\ y) \\
\mathit{IOMatch} &= \lambda x, b.(\exists f, a.x = C^2_{\mathit{Twice}}\ f\ a \wedge (\exists c.f\ a\ c \wedge f\ c\ b))
\end{aligned}$$

$$\begin{aligned}
\mathit{Apply} &= \lambda x, y, z.x = C^0_{\mathit{Add}} \wedge z = C^1_{\mathit{Add}}\ y \\
\mathit{Apply} &= \lambda x, y, z.(\exists n.x = C^1_{\mathit{Add}}\ n \wedge z = C^2_{\mathit{Add}}\ n\ y) \\
\mathit{IOMatch} &= \lambda x, b.(\exists n, a.x = C^2_{\mathit{Add}}\ n\ a \wedge a + n = b).
\end{aligned}$$

In the original P, f is used as a higher-order formal parameter, but here it has sort **closr**. The new logic program has multiple equations for *Apply* and $\mathit{IOMatch}$, which violates the rule that every top-level relational variable must be distinct. Each of the equations defines a conditional branch of a relational variable. Hence, we combine them by taking their disjunction. Specifically, suppose we have

$$\begin{aligned}
X &= \lambda x_1, \ldots, x_n.G_1 \\
X &= \lambda y_1, \ldots, y_n.G_2.
\end{aligned}$$

Then the disjunction of these two equations is given by

$$X = \lambda x_1, \ldots, x_n.(G_1 \vee G_2[x_1/y_1] \cdots [x_n/y_n]),$$

where $G_2[x_i/y_i]$ denotes the result of substituting x_i for every free occurrence of y_i in G_2. I assume that x_i does not occur bound in G_2 (or at least the substitution $[x_i/y_i]$ does not cause variable capture) for all $1 \leq i \leq n$. For readability, however, I will leave the logic program unchanged.

At this point, *Apply* is not well-sorted, since the second arguments take various sorts; e.g. **nat**, **natlist**, and **closr**. Therefore, we must create clones of *Apply*, each specializing

in a particular sort of the second argument. Let $Apply_A$ denote a clone of $Apply$ whose sort is $\mathbf{closr} \to A \to \mathbf{closr} \to o$. Similarly for $IOMatch$, I write $IOMatch_A$ for a clone of $IOMatch$ whose sort is $\mathbf{closr} \to A \to o$. Inserting these clones to appropriate places of the program, we obtain

$$
\begin{aligned}
Apply_{\mathbf{nat}} &= \lambda x, y, z.x = C^0_{Main} \wedge z = C^1_{Main}\ y \\
Apply_{\mathbf{nat}} &= \lambda x, y, z.(\exists f.x = C^1_{Twice}\ f \wedge z = C^2_{Twice}\ f\ y) \\
Apply_{\mathbf{nat}} &= \lambda x, y, z.x = C^0_{Add} \wedge z = C^1_{Add}\ y \\
Apply_{\mathbf{nat}} &= \lambda x, y, z.(\exists n.x = C^1_{Add}\ n \wedge z = C^2_{Add}\ n\ y)
\end{aligned}
$$

$$
\begin{aligned}
Apply_{\mathbf{natlist}} &= \lambda x, y, z.(\exists f.x = C^1_{TwiceMap}\ f \wedge z = C^2_{TwiceMap}\ f\ y) \\
Apply_{\mathbf{natlist}} &= \lambda x, y, z.(\exists f.x = C^1_{Map}\ f \wedge z = C^2_{Map}\ f\ y)
\end{aligned}
$$

$$
\begin{aligned}
Apply_{\mathbf{closr}} &= \lambda x, y, z.x = C^0_{TwiceMap} \wedge z = C^1_{TwiceMap}\ y \\
Apply_{\mathbf{closr}} &= \lambda x, y, z.x = C^0_{Map} \wedge z = C^1_{Map}\ y \\
Apply_{\mathbf{closr}} &= \lambda x, y, z.x = C^0_{Twice} \wedge z = C^1_{Twice}\ y
\end{aligned}
$$

$$
\begin{aligned}
IOMatch_{\mathbf{nat}} &= \lambda x, b.(\exists f, a.x = C^2_{Twice}\ f\ a \wedge (\exists c.f\ a\ c \wedge f\ c\ b)) \\
IOMatch_{\mathbf{nat}} &= \lambda x, b.(\exists n, a.x = C^2_{Add}\ n\ a \wedge a + n = b)
\end{aligned}
$$

$$
\begin{aligned}
IOMatch_{\mathbf{natlist}} &= \lambda x, ns.(\exists n.x = C^1_{Main}\ n \wedge TwiceMap\ (Add\ n)\ (\texttt{cons}\ 0\ \texttt{nil})\ ns) \\
IOMatch_{\mathbf{natlist}} &= \lambda x, ms.(\exists f, ns.x = C^2_{TwiceMap}\ f\ ns \wedge Map\ (Twice\ f)\ ns\ ms) \\
IOMatch_{\mathbf{natlist}} &= \lambda x, b.(\exists f, a.x = C^2_{Map}\ f\ a \wedge ((a = \texttt{nil} \wedge b = \texttt{nil}) \\
&\qquad \vee (\exists n, ns, m, ms.a = \texttt{cons}\ n\ ns \wedge f\ n\ m \wedge Map\ f\ ns\ ms \wedge b = \texttt{cons}\ m\ ms))).
\end{aligned}
$$

On the right hand sides of equations defining $IOMatch_A$, we have function application that involves formal higher-order parameters such as f. However, because their sorts are changed to **closr** by defunctionalization, the next step is to insert C^i_F , $Apply_A$, and $IOMatch_A$ to the right hand sides of the equations defining $IOMatch_A$. For example, the first equation of $IOMatch_{\mathbf{nat}}$ has the function application $f\ a\ c$. This is transformed into

$$\exists d.Apply_{\mathbf{nat}}\ f\ a\ d \wedge IOMatch_{\mathbf{nat}}\ d\ c.$$

Analogously, $TwiceMap\ (Add\ n)\ (\texttt{cons}\ 0\ \texttt{nil})\ ns$ in the first equation of $IOMatch_{\mathbf{natlist}}$ is transformed into

$$
\begin{aligned}
&\exists a.Apply_{\mathbf{nat}}\ C^0_{Add}\ n\ a \\
&\wedge (\exists b, c.Apply_{\mathbf{closr}}\ C^0_{TwiceMap}\ a\ b \wedge Apply_{\mathbf{natlist}}\ b\ (\texttt{cons}\ 0\ \texttt{nil})\ c \wedge IOMatch_{\mathbf{natlist}}\ c\ ns).
\end{aligned}
$$

Applying the same step to all the remaining equations that define $IOMatch_A$ in P gives

$$\begin{aligned}
IOMatch_{\mathbf{nat}} &= \lambda x, b.(\exists f, a.x = C^2_{Twice}\ f\ a \wedge (\exists c.(\exists d.Apply_{\mathbf{nat}}\ f\ a\ d \wedge IOMatch_{\mathbf{nat}}\ d\ c) \\
&\quad \wedge (\exists e.Apply_{\mathbf{nat}}\ f\ c\ e \wedge IOMatch_{\mathbf{nat}}\ e\ b))) \\
IOMatch_{\mathbf{nat}} &= \lambda x, b.(\exists n, a.x = C^2_{Add}\ n\ a \wedge a + n = b)
\end{aligned}$$

$$\begin{aligned}
IOMatch_{\mathbf{natlist}} &= \lambda x, ns.(\exists n.x = C^1_{Main}\ n \wedge (\exists a.Apply_{\mathbf{nat}}\ C^0_{Add}\ n\ a \\
&\quad \wedge (\exists b, c.Apply_{\mathbf{closr}}\ C^0_{TwiceMap}\ a\ b \wedge Apply_{\mathbf{natlist}}\ b\ (\texttt{cons}\ 0\ \texttt{nil})\ c \\
&\quad\ \wedge IOMatch_{\mathbf{natlist}}\ c\ ns))) \\
IOMatch_{\mathbf{natlist}} &= \lambda x, ms.(\exists f, ns.x = C^2_{TwiceMap}\ f\ ns \wedge (\exists a.Apply_{\mathbf{closr}}\ C^0_{Twice}\ f\ a \\
&\quad \wedge (\exists b, c.Apply_{\mathbf{closr}}\ C^0_{Map}\ a\ b \wedge Apply_{\mathbf{natlist}}\ b\ ns\ c \wedge IOMatch_{\mathbf{natlist}}\ c\ ms))) \\
IOMatch_{\mathbf{natlist}} &= \lambda x, b.(\exists f, a.x = C^2_{Map}\ f\ a \wedge ((a = \texttt{nil} \wedge b = \texttt{nil}) \\
&\quad \vee (\exists n, ns, m, ms.a = \texttt{cons}\ n\ ns \\
&\quad\ \wedge (\exists c.Apply_{\mathbf{nat}}\ f\ n\ c \wedge IOMatch_{\mathbf{nat}}\ c\ m) \\
&\quad\ \wedge (\exists d, e.Apply_{\mathbf{closr}}\ C^0_{Map}\ f\ d \wedge Apply_{\mathbf{natlist}}\ d\ ns\ e \wedge IOMatch_{\mathbf{natlist}}\ e\ ms) \\
&\quad\ \wedge b = \texttt{cons}\ m\ ms))).
\end{aligned}$$

Here we use type annotations to determine appropriate clones of *Apply* and *IOMatch* to be used. As every function application is now done through a clone of *Apply* and *IOMatch*, $Apply_A$ and $IOMatch_A$ are self-contained. Hence, we can delete all equations defining the top-level relational variables from the source program. This completes the defunctionalization of P.

Since the top-level relational variables in the original P are removed, we need to defunctionalize the goal formula component G as well. It produces

$$G' = \exists n, ns.((\exists a.Apply_{\mathbf{nat}}\ C^0_{Main}\ n\ a \wedge IOMatch_{\mathbf{natlist}}\ a\ ns) \wedge ns = \texttt{nil}).$$

To sum up, the defunctionalized logic program safety problem is $\mathcal{P}' = (\Delta', P', G')$ with the new signature $\Sigma' = (\mathbb{B}', \mathbb{S}')$, where $\mathbb{B}'$ and $\mathbb{S}'$ are given by

$$\mathbb{B}' = \mathbb{B} \cup \{\mathbf{closr}\}$$

and

$$\begin{aligned}
\mathbb{S}' = \mathbb{S} \cup \{&C^0_{Main} : \mathbf{closr}, C^1_{Main} : \mathbf{nat} \to \mathbf{closr}, \\
&C^0_{TwiceMap} : \mathbf{closr}, C^1_{TwiceMap} : \mathbf{closr} \to \mathbf{closr}, C^2_{TwiceMap} : \mathbf{closr} \to \mathbf{natlist} \to \mathbf{closr}, \\
&C^0_{Map} : \mathbf{closr}, C^1_{Map} : \mathbf{closr} \to \mathbf{closr}, C^2_{Map} : \mathbf{closr} \to \mathbf{natlist} \to \mathbf{closr}, \\
&C^0_{Twice} : \mathbf{closr}, C^1_{Twice} : \mathbf{closr} \to \mathbf{closr}, C^2_{Twice} : \mathbf{closr} \to \mathbf{nat} \to \mathbf{closr}, \\
&C^0_{Add} : \mathbf{closr}, C^1_{Add} : \mathbf{nat} \to \mathbf{closr}, C^2_{Add} : \mathbf{nat} \to \mathbf{nat} \to \mathbf{closr}\}.
\end{aligned}$$

The new sort environment is

$$\Delta' = \{Apply_{\mathbf{nat}} : \mathbf{closr} \to \mathbf{nat} \to \mathbf{closr} \to o, Apply_{\mathbf{natlist}} : \mathbf{closr} \to \mathbf{natlist} \to \mathbf{closr} \to o,$$
$$Apply_{\mathbf{closr}} : \mathbf{closr} \to \mathbf{closr} \to \mathbf{closr} \to o, IOMatch_{nat} : \mathbf{closr} \to \mathbf{nat} \to o,$$
$$IOMatch_{natlist} : \mathbf{closr} \to \mathbf{natlist} \to o\}.$$

P' consists of

$$\begin{aligned}
Apply_{\mathbf{nat}} &= \lambda x, y, z.x = C^0_{Main} \wedge z = C^1_{Main}\ y \\
Apply_{\mathbf{nat}} &= \lambda x, y, z.(\exists f.x = C^1_{Twice}\ f \wedge z = C^2_{Twice}\ f\ y) \\
Apply_{\mathbf{nat}} &= \lambda x, y, z.x = C^0_{Add} \wedge z = C^1_{Add}\ y \\
Apply_{\mathbf{nat}} &= \lambda x, y, z.(\exists n.x = C^1_{Add}\ n \wedge z = C^2_{Add}\ n\ y)
\end{aligned}$$

$$\begin{aligned}
Apply_{\mathbf{natlist}} &= \lambda x, y, z.(\exists f.x = C^1_{TwiceMap}\ f \wedge z = C^2_{TwiceMap}\ f\ y) \\
Apply_{\mathbf{natlist}} &= \lambda x, y, z.(\exists f.x = C^1_{Map}\ f \wedge z = C^2_{Map}\ f\ y)
\end{aligned}$$

$$\begin{aligned}
Apply_{\mathbf{closr}} &= \lambda x, y, z.x = C^0_{TwiceMap} \wedge z = C^1_{TwiceMap}\ y \\
Apply_{\mathbf{closr}} &= \lambda x, y, z.x = C^0_{Map} \wedge z = C^1_{Map}\ y \\
Apply_{\mathbf{closr}} &= \lambda x, y, z.x = C^0_{Twice} \wedge z = C^1_{Twice}\ y
\end{aligned}$$

$$\begin{aligned}
IOMatch_{\mathbf{nat}} &= \lambda x, b.(\exists f, a.x = C^2_{Twice}\ f\ a \wedge (\exists c.(\exists d.Apply_{\mathbf{nat}}\ f\ a\ d \wedge IOMatch_{\mathbf{nat}}\ d\ c) \\
&\qquad \wedge (\exists e.Apply_{\mathbf{nat}}\ f\ c\ e \wedge IOMatch_{\mathbf{nat}}\ e\ b))) \\
IOMatch_{\mathbf{nat}} &= \lambda x, b.(\exists n, a.x = C^2_{Add}\ n\ a \wedge a + n = b)
\end{aligned}$$

$$\begin{aligned}
IOMatch_{\mathbf{natlist}} &= \lambda x, ns.(\exists n.x = C^1_{Main}\ n \wedge (\exists a.Apply_{\mathbf{nat}}\ C^0_{Add}\ n\ a \\
&\qquad \wedge (\exists b, c.Apply_{\mathbf{closr}}\ C^0_{TwiceMap}\ a\ b \wedge Apply_{\mathbf{natlist}}\ b\ (\mathtt{cons}\ 0\ \mathtt{nil})\ c \\
&\qquad\quad \wedge IOMatch_{\mathbf{natlist}}\ c\ ns))) \\
IOMatch_{\mathbf{natlist}} &= \lambda x, ms.(\exists f, ns.x = C^2_{TwiceMap}\ f\ ns \wedge (\exists a.Apply_{\mathbf{closr}}\ C^0_{Twice}\ f\ a \\
&\qquad \wedge (\exists b, c.Apply_{\mathbf{closr}}\ C^0_{Map}\ a\ b \wedge Apply_{\mathbf{natlist}}\ b\ ns\ c \wedge IOMatch_{\mathbf{natlist}}\ c\ ms))) \\
IOMatch_{\mathbf{natlist}} &= \lambda x, b.(\exists f, a.x = C^2_{Map}\ f\ a \wedge ((a = \mathtt{nil} \wedge b = \mathtt{nil}) \\
&\qquad \vee (\exists n, ns, m, ms.a = \mathtt{cons}\ n\ ns \\
&\qquad\quad \wedge (\exists c.Apply_{\mathbf{nat}}\ f\ n\ c \wedge IOMatch_{\mathbf{nat}}\ c\ m) \\
&\qquad\quad \wedge (\exists d, e.Apply_{\mathbf{closr}}\ C^0_{Map}\ f\ d \wedge Apply_{\mathbf{natlist}}\ d\ ns\ e \wedge IOMatch_{\mathbf{natlist}}\ e\ ms) \\
&\qquad\quad \wedge b = \mathtt{cons}\ m\ ms))).
\end{aligned}$$

G' is

$$G' = \exists n, ns.((\exists a.Apply_{\mathbf{nat}}\ C^0_{Main}\ n\ a \wedge IOMatch_{\mathbf{natlist}}\ a\ ns) \wedge ns = \mathtt{nil}).$$

P' and G' are well-sorted and indeed first-order.

Suppose that the model of the background theory for $\mathcal{P}$ is A, which interprets the constant symbols from $\mathbb{S}$ in a standard way. A model A' over Σ' is then defined as follows:

- For all constant symbols inherited from Σ, A' has the same interpretation as A.
- To sort **closr**, A' assigns a universe of objects created by data constructors C_F^i.
- Functions C_F^i are interpreted in a natural way as data constructors for the algebraic data type **closr**.

The background theory for $\mathcal{P}'$ is the background theory for $\mathcal{P}$ extended with additional theorems for the **closr** universe. It is worth noting that not all functions have their respective representatives in the universe of **closr**.

$\mathcal{P}$ and $\mathcal{P}'$ have the same semantics in the sense that

$$\mathcal{P} \text{ is solvable} \iff \mathcal{P}' \text{ is solvable.}$$

This is because no relational variable is quantified in G. However, if G has quantified relational variables, there is a problem as discussed in the next section.

3.2 Quantification of higher-order variables

In the example of Section 3.1, we do not have existential quantifiers over variables of order more than 1. If we had higher-order existential quantifiers, we would have an issue with preserving the semantics (i.e. solvability) of $\mathcal{P}$.

By way of example, suppose $G = \exists f.Twice\ f\ 1\ 3$ and that P only contains $Twice$ defined as above. The resulting safety problem is not solvable, since for every prefixed point of P,

$$f = \lambda x, y.(x + 1 = y)$$

makes $Twice\ f\ 1\ 3$ hold. However, one reasonable way to defunctionalize G gives

$$G' = \exists f.(\exists a, b.Apply_{\mathbf{closr}}\ C^0_{Twice}\ f\ a \wedge Apply_{\mathbf{nat}}\ a\ 1\ b \wedge IOMatch_{\mathbf{nat}}\ b\ 3),$$

where the sort of f is now **closr**. Then there exist valuations that are prefixed points of P' but do not satisfy G'. The reason is because $Apply_A$ and $IOMatch_A$ are defined in such a way that only functions that can be created within P (i.e. partially applied functions that are represented by $C_F^i\ t_1\ \cdots t_i$, where F is a top-level relational variable) are considered by P'. As Add is not defined in P anymore, $\lambda x, y.(x + 1 = y)$ cannot arise from P. Thus, there exists a prefixed point of P' that does not satisfy G'.

Therefore, the semantics of the source safety problem are not preserved if relational variables (except for variables of sort o) are quantified in G.

In essence, my defunctionalization fails due to the fact that P' ignores any function that cannot be built from the top-level relational variables defined in P.

3.3 Elimination of higher-order quantifiers

The monotone semantics can resolve the issue with existential quantification over higher-order variables.

Monotonicity lets us eliminate higher-order existential quantifiers from all goal terms in a monotone problem. Consider $\exists_\rho \lambda x{:}\rho.F$, where ρ is a higher-order relational sort. Any function interpreted using the monotone semantics is monotone due to the use of $\Rightarrow_m$ in the definition of monotone sort frames. Hence, $\mathcal{M}[\![\lambda x{:}\rho.F]\!](\alpha)$, where $\mathtt{FV}(F) \subseteq \mathtt{dom}(\alpha)$, is a monotone function of x. If there exists $u \in \mathcal{M}[\![\rho]\!]$ such that $\mathcal{M}[\![\lambda x{:}\rho.F]\!](\alpha)(u) = 1$ for a fixed valuation α, any $v \in \mathcal{M}[\![\rho]\!]$ such that $u \subseteq_\rho v$ should satisfy $\mathcal{M}[\![\lambda x{:}\rho.F]\!](\alpha)(v) = 1$ as well. The maximum element in $\mathcal{M}[\![\rho]\!]$ is the relation that always returns 1. This is called the universal relation of sort ρ. It follows that $\mathcal{M}[\![\exists_\rho \lambda x{:}\rho.F]\!](\alpha) = \mathcal{M}[\![F[x \mapsto \lambda x_1, \ldots, x_k.\mathtt{true}]]\!](\alpha)$ for any α such that $\mathtt{FV}(F) \subseteq \mathtt{dom}(\alpha)$. Here, I assume that `true` is declared in the signature and is included in the background theory. If this is not the case, we can simply add `true` to the signature and the background theory. Henceforth, I assume that a monotone problem does not have quantifiers over higher-order relational variables.

Chapter 4

Algorithm

This chapter first formally presents the defunctionalization algorithm. It then introduces valuation extraction, which is a crucial idea in the proofs of the algorithm's completeness and soundness. The chapter concludes with proofs of completeness and soundness.

4.1 Preprocessing

Let the source monotone problem be $\mathcal{P} = (\Delta, P, G)$. Prior to defunctionalization $\mathcal{P}$, we need to eliminate all anonymous functions in P and G and then perform η-expansion to fully expand the outermost lambda abstractions defining top-level relational variables. For reasons of space, the details are not presented here. They can be found in Section A.1.

4.2 Defunctionalization algorithm

In this section, I formulate the defunctionalization algorithm via parametrised relations. A goal term to be defunctionalized is called a "source goal term", and a defunctionalized goal term is called a "target goal term". An input monotone safety problem is called a "source monotone problem" and a defunctionalized monotone safety problem is called a "target monotone problem".

One approach to formulating a defunctionalization algorithm of a functional programming language is to define a relation between source terms and target terms [Pottier and Gauthier, 2004]. I will denote the relation by $\rightsquigarrow$ and call it a transformation. $s \rightsquigarrow t$ means that s is defunctionalized into t. The word "transformation" might suggest that $\rightsquigarrow$ is not only a relation but also a function. It is in fact possible to show that $\rightsquigarrow$ returns unique outputs and hence is a function. However, it suffices to regard $\rightsquigarrow$ as a relation in this document.

Prior to presenting the core of the defunctionalization algorithm, I explain why I make use of parametrised transformations in the formulation of the algorithm.

4.2.1 Parametrised transformation

The use of parametrised transformations gives us control over variable symbols in target goal terms. To appreciate the importance of being able to specify variable symbols, consider the goal term

$$f\ x\ y,$$

where the sort of each variable is

$$\begin{aligned} f &: \mathbf{int} \to \mathbf{int} \to o \\ x &: \mathbf{int} \\ y &: \mathbf{int}. \end{aligned}$$

Further, assume $f \notin \mathtt{dom}(\Delta)$. This goal term can be defunctionalized into

$$\exists a. \mathit{Apply}_{\mathbf{int}}\ f\ x\ a \wedge \mathit{IOMatch}_{\mathbf{int}}\ a\ y, \tag{4.1}$$

where the sort of each variable is now

$$\begin{aligned} f &: \mathbf{closr} \\ x &: \mathbf{int} \\ y &: \mathbf{int} \\ a &: \mathbf{closr}. \end{aligned}$$

Note that although (4.1) is not identical to the output of the defunctionalization algorithm presented in Section 4.2.2, they are logically equivalent.

As the grammar of goal terms is defined inductively, it is natural to defunctionalize goal terms inductively. (4.1) consists of two components:

$$\begin{aligned} &\mathit{Apply}_{\mathbf{int}}\ f\ x\ a \\ &\mathit{IOMatch}_{\mathbf{int}}\ a\ y. \end{aligned}$$

The former corresponds to the partial application of f to x and the latter corresponds to the application of $(f\ x)$ to y. Hence, the structure of (4.1) roughly reflects the structure of $f\ x\ y$, where the curried function f is applied to x first and then to y. However, the two components in (4.1) cannot be separated cleanly. The problem is that both components refer to the same quantified variable a. Hence, it is necessary to establish a "communication channel" between the defunctionalization of $f\ x$ and the defunctionalization of $(f\ x)\ y$. More specifically, we need to either specify what variable should be used in the first component or inspect it and then copy the variable symbol used in it to the second component.

As it is certainly not straightforward to define helper functions that extract variable symbols from target goal terms, I opted to pass variable symbols to target goal terms. One approach is to use contexts. For the above example, we can defunctionalize $f\ x$ into a context $\mathit{Apply}_{\mathbf{int}}\ f\ x\ X$, where X is a hole. We can then substitute a concrete variable symbol into X.

Another approach to passing variable symbols is to use parametrised transformations. If a parametrised transformation $\rightsquigarrow^X$ is defined in such a way that $(f\ x) \rightsquigarrow^X (Apply_{\mathbf{int}}\ f\ x\ X)$ holds, we can invoke $\rightsquigarrow^X$ with a specific variable symbol substituted into X.

The transformation in the context-based approach sometimes returns contexts and other times returns goal terms with no holes. Since this can be confusing to readers, I adopted the approach based on parametrised transformations.

4.2.2 Defunctionalization

Formal presentation Let $\mathcal{P} = (\Delta, P, G)$ be the source monotone problem that has already been preprocessed by the procedure explained in Section 4.1. Due to the preprocessing, the formal parameters of each outermost lambda abstraction defining a top-level relational variable are visible. Also, P and G contain no anonymous functions. Given $P = \lambda x_1, \ldots, x_m.F$, where F is not a lambda abstraction, let us call F the body of P. Because $G : o$ and hence is not a function, the body of G is G itself. Since lambda abstractions only appear at the top level of syntax trees of P and G, the bodies of P and G are free of lambda abstractions. The defunctionalization of the bodies is guided by the following inference rules:

$$\frac{c \in \{\wedge, \vee\} \qquad E \rightsquigarrow E' \qquad F \rightsquigarrow F'}{(c\ E\ F) \rightsquigarrow (c\ E'\ F')}\ (\textsc{LogSym}) \qquad \frac{F \rightsquigarrow F'}{\exists_b x.F \rightsquigarrow \exists_b x.F'}\ (\textsc{Exi})$$

$$\frac{\mathrm{head}(E) \notin \{\wedge, \vee\} \qquad \Delta \vdash (E\ F) : \rho \qquad \rho \neq o \qquad (E\ F) \rightsquigarrow^X_A H}{(E\ F) \rightsquigarrow^X H}\ (\textsc{App})$$

$$\frac{\mathrm{head}(E) \notin \{\wedge, \vee\} \qquad \Delta \vdash (E\ F) : o \qquad (E\ F) \rightsquigarrow_M H}{(E\ F) \rightsquigarrow H}\ (\textsc{Match})$$

$$\frac{E \rightsquigarrow^x E' \qquad \Delta \vdash F : \sigma \qquad \sigma \rightsquigarrow_T \sigma \qquad F \rightsquigarrow F'}{(E\ F) \rightsquigarrow^X_A \exists_{\mathbf{closr}} x.(E' \wedge Apply_\sigma\ x\ F'\ X)}\ (\textsc{App-Base})^*$$

$$\frac{E \rightsquigarrow^x E' \qquad \Delta \vdash F : \sigma \qquad \sigma \rightsquigarrow_T \mathbf{closr} \qquad F \rightsquigarrow^y F'}{(E\ F) \rightsquigarrow^X_A \exists_{\mathbf{closr}} x.(E' \wedge \exists_{\mathbf{closr}} y.(F' \wedge Apply_{\mathbf{closr}}\ x\ y\ X))}\ (\textsc{App-Arrow})^*$$

$$\frac{E \rightsquigarrow^x E' \qquad \Delta \vdash F : \sigma \qquad \sigma \rightsquigarrow_T \sigma \qquad F \rightsquigarrow F'}{(E\ F) \rightsquigarrow_M \exists_{\mathbf{closr}} x.(E' \wedge IOMatch_\sigma\ x\ F')}\ (\textsc{Match-Base})^*$$

$$\frac{E \rightsquigarrow^x E' \qquad \Delta \vdash F : \sigma \qquad \sigma \rightsquigarrow_T \mathbf{closr} \qquad F \rightsquigarrow^y F'}{(E\ F) \rightsquigarrow_M \exists_{\mathbf{closr}} x.(E' \wedge \exists_{\mathbf{closr}} y.(F' \wedge IOMatch_{\mathbf{closr}}\ x\ y))}\ (\textsc{Match-Arrow})^*$$

$$\frac{\varphi \in Fm \cup Tm}{\varphi \rightsquigarrow \varphi}\ (\textsc{ConstrLan}) \qquad \frac{x \text{ is a variable} \qquad x : o}{x \rightsquigarrow x}\ (\textsc{Var-Base})$$

$$\frac{x \text{ is a variable} \qquad x : \rho \qquad \rho \neq o}{x \rightsquigarrow^X X = x}\ (\textsc{Var-Arrow}) \qquad \frac{x \in \Delta}{x \rightsquigarrow^X X = C_x^0}\ (\textsc{TopVar})$$

$$\frac{\texttt{order}(b) = 1}{b \rightsquigarrow_T b}\ (\textsc{Base}) \qquad \frac{\texttt{order}(\tau) > 1}{\tau \rightsquigarrow_T \textbf{closr}}\ (\textsc{Arrow})$$

The function head is defined by

$$\begin{aligned} \text{head}(x) &= x && x \text{ is a variable} \\ \text{head}(c) &= c && c \in \{\wedge, \vee\} \\ \text{head}(\varphi) &= \varphi && \varphi \in Fm \cup Tm \\ \text{head}(E) &= E && E \text{ is in the form } \lambda x.F \text{ or } \exists x.F \\ \text{head}(E\ F) &= \text{head}(E). \end{aligned}$$

This function returns the head symbols of goal terms.

In the conclusions of the rules whose names are marked with *, quantified variables x and y are assumed to be different from any variable symbol occurring in E' and F' before substitutions $[X \mapsto x]$ and $[X \mapsto y]$ are applied.

Transformation types The abbreviations of the rules' names and what they stand for are summarised below:

Abbreviation	Full form
LogSym	Logical constant symbols
Exi	Existential quantifier
App	Apply
Match	IOMatch
App-Base	Apply for a base sort
App-Arrow	Apply for an arrow sort
Match-Base	IOMatch for a base sort
Match-Arrow	IOMatch for an arrow sort
ConstrLan	Constraint language
Var-Base	Variable of base sort
Var-Arrow	Variable of an arrow sort
TopVar	Top-level relational variable
Base	Base sort
Arrow	Arrow sort

In the above inference rules, we have five types of transformations: $\rightsquigarrow$, $\rightsquigarrow^X$, $\rightsquigarrow^X_A$, $\rightsquigarrow_M$, and $\rightsquigarrow_T$. Superscripts of $\rightsquigarrow$ store parameters, and subscripts denote the classes of the transformations. 'A' in $\rightsquigarrow^X_A$ is short for Apply, 'M' in $\rightsquigarrow_M$ is short for Match, and 'T' in $\rightsquigarrow_T$ is short for Types (i.e. sorts).

The transformations $\rightsquigarrow$ and $\rightsquigarrow^X$ are applied to goal terms that contain no lambda abstractions. They are used to transform the bodies of lambda abstractions defining top-level relational variables. In $\rightsquigarrow^X$, X is a parameter into which a variable symbol

is substituted. Hence, $\rightsquigarrow^X$ is a parametrised relation that returns an appropriate goal term according to the parameter X passed to the relation.

The transformation $\rightsquigarrow_A^X$ is for function application that produces goal terms with arrow sorts. Because the result of the function application is not of base sort, the argument of the function application cannot be the last parameter of any relational variable (otherwise, the result of the function application would have the sort o). Hence, $\rightsquigarrow_A^X$ replaces such function application with an instance of *Apply*. Like $\rightsquigarrow^X$, $\rightsquigarrow_A^X$ has a parameter for variable symbols. Notice that X in the inference rules defining $\rightsquigarrow^X$ and $\rightsquigarrow_A^X$ acts as a "metavariable" and hence is used as a "pattern" in pattern matching.

In contrast to $\rightsquigarrow_A^X$, $\rightsquigarrow_M$ is for function application that produces goal terms of base sort. Such function application is replaced with an instance of $IOMatch$.

Lastly, $\rightsquigarrow_T$ transforms sorts.

4.2.3 Bindings of variables and quantifiers

In the four inference rules marked with *, new quantified variables are introduced in the result of transformation. To avoid variable capture and disambiguate bindings of quantifiers and quantified variables, the newly introduced quantified variables must be distinct from all variable symbols occurring in E' and F' before we apply substitutions $[X \mapsto x]$ and $[X \mapsto y]$. Hence, it is necessary to calculate E' and F' before we can select suitable symbols for the quantified variables in the four rules' conclusions.

To illustrate the need for using fresh variables, consider the partially applied goal term

$$Add\ x, \tag{4.2}$$

where the sort of each variable is

$$\begin{aligned} Add &: \mathbf{int} \to \mathbf{int} \to \mathbf{int} \to o \\ x &: \mathbf{int}. \end{aligned}$$

Further, assume $Add \in \Delta$. By (TopVar), we have

$$Add \rightsquigarrow^X X = C^0_{Add}, \tag{4.3}$$

where X is to be specified when the enclosing goal term is defunctionalized. We also have

$$x \rightsquigarrow x$$

by (Var-Base).

To produce a target term of $Add\ x$, we apply (App) and (App-Base). If x is used for a new quantified variable, we obtain

$$Add\ x \rightsquigarrow^X \exists_{\mathbf{closr}} x.x = C^0_{Add} \wedge Apply_{\mathbf{int}}\ x\ x\ X. \tag{4.4}$$

Variable capture happens in (4.4) since the second argument of $Apply_{\mathbf{int}}$, which ought to be a free variable, is now bound by $\exists_{\mathbf{closr}}$. To disambiguate this expression, a fresh variable symbol y is used for the quantified variable:

$$Add\ x \rightsquigarrow^X \exists_{\mathbf{closr}} y.y = C^0_{Add} \wedge Apply_{\mathbf{int}}\ y\ x\ X. \tag{4.5}$$

X in the parametrised transformations specifies what variable symbol is used to denote the entity of sort **closr** that represents the source goal term.

For example, consider $Add\ x$ in (4.2). In the target term of Add in (4.3), X denotes the entity of sort **closr** that represents Add, i.e. C^0_{Add}, because X and C^0_{Add} are connected by equality.

Also, in the target term of $Add\ x$ in (4.5), we have $Apply_{\mathbf{int}}\ y\ x\ X$, where $y = C^0_{Add}$. The third parameter of $Apply$, which always has the sort **closr**, represents the result of applying the function represented by the first parameter to the entity represented by the second parameter. Hence, X denotes a closure that represents $Add\ x$.

4.2.4 Target monotone problems

The defunctionalized monotone problem is given by $\mathcal{P}' = (\Delta', P', G')$ with a new signature $\Sigma' = (\mathbb{B}', \mathbb{S}')$, where $\mathbb{B}'$ and $\mathbb{S}'$ are given by

$$\begin{aligned}
\mathbb{B}' &= \mathbb{B} \cup \{\mathbf{closr}\} \\
\mathbb{S}' &= \mathbb{S} \cup \{(=_{\mathbf{closr}}) : \mathbf{closr} \to \mathbf{closr} \to o\} \\
&\quad \cup \{C^i_X : \sigma'_1 \to \cdots \to \sigma'_i \to \mathbf{closr} \mid X : \sigma_1 \to \cdots \to \sigma_m \to o \in \Delta, \\
&\qquad 0 \le i < m, \sigma_j \rightsquigarrow_T \sigma'_j \text{ for all } 1 \le j \le i\}.
\end{aligned}$$

Here, **closr**, $(=_{\mathbf{closr}})$, and C^i_X are assumed to be fresh. The new sort environment Δ' is

$$\begin{aligned}
\Delta' = \{&Apply_A : \mathbf{closr} \to A \to \mathbf{closr} \to o \mid A \in \mathbb{B}'\} \\
&\cup \{IOMatch_A : \mathbf{closr} \to A \to o \mid A \in \mathbb{B}'\}.
\end{aligned} \tag{4.6}$$

Some of $Apply_A$ and $IOMatch_A$ may be redundant. P' is defined as

$$P' = P'_{\text{Apply}} \cup P'_{\text{IOMatch}}, \tag{4.7}$$

where P'_{Apply} and P'_{IOMatch} are

$$\begin{aligned}
P'_{\text{Apply}} = \{&Apply_{\sigma'_{n+1}} = \lambda x, y, z.(\exists a_1, \ldots, a_n.x = C^n_X\ a_1\ \cdots\ a_n \wedge z = C^{n+1}_X\ a_1\ \cdots\ a_n\ y) \\
&\mid (X = \lambda x_1{:}\sigma_1, \ldots, x_m{:}\sigma_m.F) \in P, \mathbf{ar}(X) = m, 0 \le n \le m-2, \sigma_{n+1} \rightsquigarrow_T \sigma'_{n+1}\}
\end{aligned} \tag{4.8}$$

$$\begin{aligned}
P'_{\text{IOMatch}} = \{&IOMatch_{\sigma'_m} = \lambda x, x_m.(\exists x_1, \ldots, x_{m-1}.x = C^{m-1}_X\ x_1\ \cdots\ x_{m-1} \wedge F') \\
&\mid (X = \lambda x_1{:}\sigma_1, \ldots, x_m{:}\sigma_m.F) \in P, \mathbf{ar}(X) = m, \sigma_m \rightsquigarrow_T \sigma'_m, F \rightsquigarrow F'\}
\end{aligned} \tag{4.9}$$

Note that F in (4.9) must have the sort o. This indeed holds and follows from Lemma D.3.1). In the definition of P'_{IOMatch}, every occurrence of $x_1, \ldots, x_m$ in F' is bound by the outermost $\exists$ in the body of $IOMatch_{\sigma'_m}$.

The defunctionalized goal formula G' is given by

$$G \rightsquigarrow G'.$$

Lastly, the constraint language, particularly the background theory, for $\mathcal{P}'$ need to be defined. Let the constraint language for $\mathcal{P}$ be (Tm, Fm, Th) and the constraint language for $\mathcal{P}'$ be (Tm', Fm', Th'). Tm' and Fm' are informally defined as extensions of Tm and Fm with terms and formulas containing $(=_{\mathbf{closr}})$ and C^i_X for some relational variable $X \in \Delta$. Because formal definitions of Tm' and Fm' are not critical to the proofs of the defunctionalization algorithm's correctness, I will not formally define them.

The background theory Th can always be characterized by a set of structures S such that $F \in Th$ if and only $A \models F$ for all $A \in S$. For each $A \in S$, a structure A' for $\mathcal{P}'$ is defined as follows:

- To all sorts inherited from $\mathbb{B}$, A' assigns the same universe as A.
- To the sort **closr**, A' assigns the universe of objects that can be constructed by the data constructors $C^i_X \in \mathbb{S}' \setminus \mathbb{S}$. Informally, the universe assigned to **closr** is

  $$\begin{aligned} A'_{\mathbf{closr}} = \{(X, t_1, \ldots, t_k) \mid\ & X : \sigma_1 \to \cdots \to \sigma_m \to o \in \Delta, 0 \leq k < m, \\ & \sigma_i \rightsquigarrow_T \sigma'_i \text{ for each } 1 \leq i \leq k, t_i \in A'_{\sigma'_i} \text{ for each } 1 \leq i \leq k\}. \end{aligned}$$

  In other words, the universe is the set of tuples in which the first component denotes a top-level relational variable and the remaining components represent the actual parameters that have been supplied to the relational variable. This definition is informal because we may have $A'_{\sigma'_i} = A'_{\mathbf{closr}}$; i.e. the definition may be circular. In that case, the definition does not qualify as a formal definition. Another problem we have with this informal definition is that infinitely nested tuples are admitted. To get around this issue, I provide an inference rule to construct elements in $A'_{\mathbf{closr}}$:

  $$\frac{X : \sigma_1 \to \cdots \to \sigma_m \to o \in \Delta \qquad 0 \leq k < m \qquad t_i \in A'_{b_i} \text{ for each } 1 \leq i \leq m}{(X, t_1, \ldots, t_k) \in A'_{\mathbf{closr}}}$$

  where $\sigma_i \rightsquigarrow_T b_i$ for each $1 \leq i \leq k$.
- For all constant symbols inherited from $\mathbb{S}$, A' interprets them in the same way as A.
- The interpretation of $(=_{\mathbf{closr}}) : A'_{\mathbf{closr}} \to A'_{\mathbf{closr}} \to \mathbb{2}$ is determined by this inference rule

  $$\frac{X : \sigma_1 \to \cdots \to \sigma_k \to o \in \Delta \qquad 0 \leq k < m \qquad t_i =_{b_i} s_i \text{ for each } 1 \leq i \leq k}{(X, t_1, \cdots, t_k) =_{\mathbf{closr}} (X, s_1, \cdots, s_k)}$$

  where $\sigma_i \rightsquigarrow_T b_i$ for each $1 \leq i \leq k$. The interpretation of $(=_{\mathbf{closr}})$ is well-defined since the equality $(=_b)$ for each $b \in \mathbb{B}$ exists.

- C^i_X is interpreted as a function that takes in i many arguments and returns an appropriate object from $A'_{\mathbf{closr}}$. Formally, it is defined by

$$C^i_X(t_1, \ldots, t_i) = (X, t_1, \ldots, t_i).$$

The new background theory for the target monotone problem is obtained by extending each model in S:

$$S' = \{A' \mid A \in S\},$$

where S' is a set of models characterizing Th'.

4.3 Valuation extraction

Given a monotone problem $\mathcal{P} = (\Delta, P, G)$, a model of P is an element of $\mathcal{M}[\![\Delta]\!]$ such that it is a prefixed point of the one-step consequence operator $T^{\mathcal{M}}_{P:\Delta}$. If α is a model of P and $X \in \mathtt{dom}(\Delta)$, I will call $\alpha(X)$ a model of X. $\mathcal{P}$ is said to be solvable if for every model of the background theory, there exists a model α of P such that $\mathcal{M}[\![G]\!](\alpha) = 0$. I will call such a model of P a solution to $\mathcal{P}$.

To prove completeness of the algorithm, my approach is to extract a solution to the target monotone problem from a solution to the source monotone problem. Hence, I will start with explaining how valuations can be extracted.

In this section, a source monotone problem is assumed to have been preprocessed. As it is relatively easy to see that the preprocessing step preserves semantics, a formal proof for that will not be provided.

4.3.1 Demonstration

To illustrate how extraction works, consider a source monotone problem $\mathcal{P} = (\Delta, P, G)$ with the first-order signature being $\Sigma = (\mathbb{B}, \mathbb{S})$, where

$$\mathbb{B} = \{\mathbf{nat}, o\}$$
$$\mathbb{S} = \{+ : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}, (=) : \mathbf{nat} \to \mathbf{nat} \to o\} \cup \{n : \mathbf{nat} \mid n \in \mathbb{N}\}.$$

P contains

$$Add = \lambda a, b, c.a + b = c$$
$$Twice = \lambda f, a, b.(\exists c.f\ a\ c \wedge f\ c\ b).$$

The sort environment for these two top-level relational variables is

$$\Delta = \{Add : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat} \to o, Twice : (\mathbf{nat} \to \mathbf{nat} \to o) \to \mathbf{nat} \to \mathbf{nat} \to o\}.$$

Since G is irrelevant to the discussion of how to extract solutions, it is unnecessary to specify G. Let A be the structure that assigns the universe of natural numbers, denoted by $\mathbb{N}$, to the sort $\mathbf{nat}$. A interprets the symbols in $\mathbb{S}$ as they are.

I will consider a specific model α of P that is defined by

$$\alpha : Add \mapsto add \qquad \alpha : Twice \mapsto twice.$$

The model of Add is $add : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{2}$ defined as

$$add\ a\ b\ c = \begin{cases} 1 & \text{if } a + b = c \\ 0 & \text{otherwise.} \end{cases}$$

The model of $Twice$ is $twice : (\mathbb{N} \to \mathbb{N} \to \mathbb{2}) \to \mathbb{N} \to \mathbb{N} \to \mathbb{2}$ is defined as

$$twice\ f\ a\ b = \begin{cases} 1 & \text{if } \exists c.f\ a\ c = 1 \wedge f\ c\ b = 1 \\ 0 & \text{otherwise.} \end{cases}$$

In fact, regardless of valuation α, add coincides with $\mathcal{M}[\![\Delta \vdash (\lambda a, b, c.a + b = c) : o]\!](\alpha)$, and similarly $twice$ coincides with $\mathcal{M}[\![\Delta \vdash (\lambda f, a, b.(\exists c.f\ a\ c \wedge f\ c\ b)) : o]\!](\alpha)$. In other words, any model of P is larger than or equal to α; hence, α is the least model of P.

$\mathcal{P}'$ is defunctionalized into $\mathcal{P}' = (\Delta', P', G')$, where P' contains

$$\begin{aligned} Apply_{\mathbf{nat}} &= \lambda x, y, z.x = C^0_{Add} \wedge z = C^1_{Add}\ y \\ Apply_{\mathbf{nat}} &= \lambda x, y, z.(\exists n.x = C^1_{Add}\ n \wedge z = C^2_{Add}\ n\ y) \\ IOMatch_{\mathbf{nat}} &= \lambda x, c.(\exists n, a.x = C^2_{Add}\ a\ b \wedge a + b = c) \end{aligned}$$

$$\begin{aligned} Apply_{\mathbf{closr}} &= \lambda x, y, z.x = C^0_{Twice} \wedge z = C^1_{Twice}\ y \\ Apply_{\mathbf{nat}} &= \lambda x, y, z.(\exists f.x = C^1_{Twice}\ f \wedge z = C^2_{Twice}\ f\ y) \\ IOMatch_{\mathbf{nat}} &= \lambda x, b.(\exists f, a.x = C^2_{Twice}\ f\ a \wedge (\exists c.(\exists d.Apply_{\mathbf{nat}}\ f\ a\ d \wedge IOMatch_{\mathbf{nat}}\ d\ c) \\ &\qquad \wedge (\exists e.Apply_{\mathbf{nat}}\ f\ c\ e \wedge IOMatch_{\mathbf{nat}}\ e\ b))). \end{aligned}$$

I will now work out a model of P' induced by α. The universe of **nat** in P' remains $\mathbb{N}$. The universe of **closr** in P', denoted by $A'_{\mathbf{closr}}$, is constructed by

$$\frac{X : \sigma_1 \to \cdots \to \sigma_m \to o \in \Delta \qquad 0 \le k < m \qquad t_i \in A'_{b_i} \text{ for each } 1 \le i \le m}{(X, t_1, \ldots, t_k) \in A'_{\mathbf{closr}}}$$

where $\sigma_i \rightsquigarrow_T b_i$ for each $1 \le i \le k$. A model for $Apply_{\mathbf{nat}}$ is the function $apply_{\mathbf{nat}} : A'_{\mathbf{closr}} \to \mathbb{N} \to A'_{\mathbf{closr}} \to \mathbb{2}$ defined as

$$apply_{\mathbf{nat}}\ m_1\ n\ m_2 = \begin{cases} 1 & \text{if } m_1 = C^0_X \wedge m_2 = C^1_X\ n \\ & \text{or } \exists n_1.(m_1 = C^1_X\ n_1 \wedge m_2 = C^2_X\ n_1\ n) \\ 0 & \text{otherwise,} \end{cases}$$

where $X \in \{Add, Twice\}$.

More generally, the model for $Apply_B$ is a function $apply_B : A'_{\mathbf{closr}} \to A'_B \to A'_{\mathbf{closr}} \to \mathbb{2}$ that takes three inputs: m_1, n, m_2. Here, the universes A'_b, where $b \in \mathbb{B} \cup \{\mathbf{closr}\}$,

are defined in Section 4.2.4. The parameter m_1 is a closure that represents a partially applied function, and n is an input to be augmented to m_1. Thus, $apply_B$ is defined as

$$apply_B\ m_1\ n\ m_2 = \begin{cases} 1 & \text{if } m_2 =_{\mathbf{closr}} \text{append}(m_1, n) \\ 0 & \text{otherwise,} \end{cases}$$

where $\text{append} : A'_{\mathbf{closr}} \to A'_B \to A'_{\mathbf{closr}}$ is

$$\text{append}((X, t_1, \ldots, t_k), t_{k+1}) = (X, t_1, \ldots, t_k, t_{k+1}).$$

A model for $Apply_{\mathbf{closr}}$ in P' is therefore given by $apply_{\mathbf{closr}}$.

Next, I consider $IOMatch_{\mathbf{nat}}$. Because it has two branches corresponding to different top-level relational variables from the source problem, I will derive a model for each branch separately. These two models will be merged later to form a single model for $IOMatch_{\mathbf{nat}}$.

The first branch of $IOMatch_{\mathbf{nat}}$ is obtained by defunctionalizing Add. This branch has the role of determining whether the first input, which should be a closure of Add, can be evaluated to the second input. The first input is expected to have the form (Add, n_1, n_2). Hence, the model for the first branch of $IOMatch_{\mathbf{nat}}$ is the function $add' : A'_{\mathbf{closr}} \to \mathbb{N} \to \mathbb{2}$ defined as

$$add'\ m\ n = \begin{cases} 1 & \text{if } m = (Add, n_1, n_2), n = n_1 + n_2 \\ 0 & \text{otherwise.} \end{cases}$$

Notice that this function is similar to add in that both of them perform addition and return 1 whenever the last input matches the result of addition. They only differ in the representation of the inputs: in add, two numbers to be summed are stored in the first two parameters, whereas in add', they are stored inside the closure in the first parameter. Capturing the similarity between add and add', we can easily formalize how to convert a model for a top-level relational variable to a model for the corresponding $IOMatch$ branch when **closr** is not involved.

Next, I work out how to interpret the second branch of $IOMatch_{\mathbf{nat}}$, which is obtained by defunctionalizing $Twice$. Elements of base sort in $\mathcal{P}'$ can be converted to corresponding elements in $\mathcal{P}$ as follows:

$$\begin{aligned} \text{expand}_\alpha(t) &= t && \text{if } t \text{ is not of sort } \mathbf{closr} \\ \text{expand}_\alpha((X, t_1, \ldots, t_k)) &= \alpha(X)\ \text{expand}(t_1)\ \cdots\ \text{expand}(t_k) && \text{otherwise.} \end{aligned}$$

Using the expand_α function, I define $twice' : A'_{\mathbf{closr}} \to \mathbb{N} \to \mathbb{2}$ as

$$twice'\ m\ n = \begin{cases} 1 & \text{if } m = (Twice, f, n_1) \wedge twice\ \text{expand}_\alpha(f)\ n_1\ n = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Alternatively, $twice\ \text{expand}_\alpha(f)\ n_1\ n$ can be written as $\text{expand}(m)\ \text{expand}(n)$. The interpretation of the second branch of $IOMatch_{\mathbf{nat}}$ is $twice'$.

By construction, $twice'\ (Twice, f, n_1)\ n_2 = 1$ implies $twice$ expand$(f)\ n_1\ n_2 = 1$. However, the converse does not hold. For example, $twice'\ (Twice, (Add, 1), 0)\ 2$ holds, as does $twice\ (add\ 1)\ 0\ 2$. By contrast, $twice\ (\lambda a, b.a - 1 = b)\ 2\ 0$ holds, whilst $twice'\ t\ 2\ 0$ does not hold for any t of sort **closr**. This is owing to the fact that $A'_{\mathbf{closr}}$ only contains closures representing partially applied functions that are expressible using the top-level relational variables in P.

As the two branches of $IOMatch_{\mathbf{nat}}$ are (syntactically) combined by taking their disjunction, the interpretation of $IOMatch_{\mathbf{nat}}$ is obtained by taking the disjunction of its constituent interpretations. Hence, the resulting model for $IOMatch_{\mathbf{nat}}$ is $iomatch_{\mathbf{nat}} : A'_{\mathbf{closr}} \to \mathbb{N} \to \mathbb{2}$ defined as

$$iomatch_{\mathbf{nat}}\ m\ n = add'\ m\ n \vee twice'\ m\ n.$$

This can be made more general:

$$iomatch_{\mathbf{nat}}\ m\ n = \text{expand}_\alpha(m)\ \text{expand}_\alpha(n).$$

4.3.2 Formalization of valuation extraction

Given a first-order signature $\Sigma = (\mathbb{B}, \mathbb{S})$, suppose that a source problem is $\mathcal{P} = (\Delta, P, G)$. Assume that the target problem of $\mathcal{P}$ is $\mathcal{P}' = (\Delta', P', G')$ and that the new signature is $\Sigma' = (\mathbb{B}', \mathbb{S}')$, where $\mathbb{B}' = \mathbb{B} \cup \{\mathbf{closr}\}$. The derivation of $\mathbb{S}'$ is presented in Section 4.2.4.

Let A be a Σ-structure used to interpret $\mathcal{P}$ and A' be the structure for $\mathcal{P}'$ obtained from A as explained in Section 4.2.4. I write A'_B for the universe assigned to $B \in \mathbb{B}'$ by A'. Also, assume that α is a valuation drawn from $\mathcal{M}[\![\Delta]\!]$. I will now explain how to derive a valuation $\alpha' \in \mathcal{M}[\![\Delta']\!]$ from α.

Each top-level relational variable in Δ' is either $Apply_B$ or $IOMatch_B$, where $B \in \mathbb{B}'$.

As for $Apply_B$, α' maps it to $apply_B : A'_{\mathbf{closr}} \to A'_B \to A'_{\mathbf{closr}} \to \mathbb{2}$ defined as

$$apply_B\ m_1\ n\ m_2 = \begin{cases} 1 & \text{if } m_2 =_{\mathbf{closr}} \text{append}(m_1, n) \\ 0 & \text{otherwise,} \end{cases}$$

where append : $A'_{\mathbf{closr}} \to A'_B \to A'_{\mathbf{closr}}$ is

$$\text{append}((X, t_1, \ldots, t_k), t_{k+1}) = (X, t_1, \ldots, t_k, t_{k+1}).$$

With respect to $IOMatch_B$, its interpretation is given by $iomatch_B : A'_{\mathbf{closr}} \to A'_B \to \mathbb{2}$ defined as

$$iomatch_B\ m\ n = \text{expand}_\alpha(m)\ \text{expand}_\alpha(n)$$

The function expand$_\alpha$ is defined as

$$\begin{aligned} \text{expand}_\alpha(s) &= s && \text{if } s : b, b \in \mathbb{B} \\ \text{expand}_\alpha((Y, s_1, \ldots, s_l)) &= \alpha(Y)\ \text{expand}_\alpha(s_1)\ \cdots\ \text{expand}_\alpha(s_l) && \text{otherwise.} \end{aligned}$$

If $iomatch_B\ m\ n = \text{expand}_\alpha(m)\ \text{expand}_\alpha(n)$ is not well-defined due to type mismatch, then it is set to 0.

The valuation α' is therefore

$$\begin{aligned}\alpha' = &\{(Apply_B, apply_B) \mid B \in \mathbb{B}'\}\} \\ &\cup \{(IOMatch_B, iomatch_B) \mid B \in \mathbb{B}'\}\}.\end{aligned}$$

Henceforth, I will write $\alpha' = T_f(\alpha)$ to mean that α' is derived from α by the above procedure, where α is a valuation for P.

4.3.3 Monotonicity of α'

We need to check whether the model of each $X \in \texttt{dom}(\Delta')$ assigned by α' is monotone. In fact, α' is "nearly" monotone but is not truly monotone, since $\alpha'(Apply_o)$ is not monotone. Appendix B describes how to get around this issue.

4.4 Meaning preservation

Meaning preservation means the preservation of source problems' semantics. Hence, meaning preservation is achieved when target monotone problems are solvable if and only if source monotone problems are solvable.

4.4.1 First direction

In this section, I prove that it is possible to produce a solution to the target monotone problem from a solution to the source monotone problem.

As usual, given a first-order signature $\Sigma = (\mathbb{B}, \mathbb{S})$, suppose that a source problem is $\mathcal{P} = (\Delta, P, G)$. Assume that $\mathcal{P}$ is defunctionalized into $\mathcal{P}' = (\Delta', P', G')$ and that its new signature is $\Sigma' = (\mathbb{B}', \mathbb{S}')$.

Let Th be a background theory for $\mathcal{P}$ and Th' be the background theory for $\mathcal{P}'$ derived from Th. Assume $A \in Th$ and $A' \in Th'$, where A' is built from A as presented in Section 4.2.4.

First, I establish the relationship between the semantics of source goal terms and semantics of target goal terms. A source goal term has either an arrow sort or a base sort. I will first illustrate the connection between the semantics of source and target goal terms in the case when the source goal terms are of arrow sorts.

If $s \rightsquigarrow^X t$ and s has a relational arrow sort, X can be thought of as a variable (more precisely, a placeholder/hole for a variable) of sort **closr** that represents s. For instance, the partially applied function $Add\ 1$ is defunctionalized into

$$\exists_{\mathbf{closr}} x.(x = C^0_{Add} \wedge Apply_{\mathbf{nat}}\ x\ 1\ X), \tag{4.10}$$

where X is to be specified by a defunctionalization step at a higher level. In (4.10), X appears in the last parameter of $Apply_{\mathbf{nat}}$. Hence, X can be considered as a variable

that represents the result of applying Add, which is represented by $x = C^0_{Add}$, to 1. To put it differently,

$$\mathcal{M}[\![\exists_{\mathbf{closr}} x.(x = C^0_{Add} \wedge Apply_{\mathbf{nat}}\ x\ 1\ X)]\!]([X \mapsto (Add, 1)]) = 1;$$

that is, (4.10) holds when we substitute $X = C^1_{Add}\ 1$. Moreover, it is worth observing that

$$\text{expand}_\alpha((Add, 1)) = \mathcal{M}[\![Add\ 1]\!](\alpha),$$

where α is a valuation of the source goal term $Add\ 1$.

On the other hand, if $\Gamma \vdash s : b$, where $b \in \mathbb{B}$, and $s \rightsquigarrow t$, then s and t have the same semantics. For example, consider $s = Add\ 1\ 1\ 2$ that is to be interpreted using the valuation $\alpha = [Add \mapsto add]$. s is defunctionalized into

$$\exists_{\mathbf{closr}} x, y, z.\ \left(x = C^0_{Add} \wedge Apply_{\mathbf{nat}}\ x\ 1\ y \wedge Apply_{\mathbf{nat}}\ y\ 1\ z \wedge IOMatch_{\mathbf{nat}}\ z\ 2\right).$$

Let this target goal term be denoted by t. t is interpreted using α', which is obtained by applying the procedure presented in Section 4.3.2. It gives $\alpha' = [Apply_{\mathbf{nat}} \mapsto apply_{\mathbf{nat}}, IOMatch_{\mathbf{nat}} \mapsto add']$.

Now we have

$$\mathcal{M}[\![s]\!](\alpha) = \mathcal{M}[\![t]\!](\alpha')$$

since both sides of the equation evaluate to 1.

To express this formally, consider a well-sorted source goal term s over Σ that contains no lambda abstractions. Suppose the following:

- s is a goal term (or a subgoal term) from P. The structure A is used to interpret s.
- $\Gamma \vdash s : \sigma$, where σ is either a relational arrow sort or a base sort. Because s is well-sorted, $\texttt{FV}(s) \subseteq \texttt{dom}(\Gamma)$.
- $s \rightsquigarrow^X t$. The structure A' is used to interpret t. In s, both ordinary variables and top-level relational variables are treated as variables. However, in t, ordinary variables from s have the same status, whilst top-level relational variables from s become constant symbols in t.
- $\Gamma' \vdash t : \sigma'$, where $\sigma \rightsquigarrow_T \sigma'$ and $\texttt{FV}(t) \subseteq \Gamma'$. Γ' can be equal to

  $$\begin{aligned}&\{u : \sigma' \mid u \in \texttt{FV}(s) \setminus \texttt{dom}(\Delta), u : \sigma \in \Gamma\}\\ &\cup \{Apply_B : \mathbf{closr} \to B \to \mathbf{closr} \to o \mid B \in \mathbb{B}'\}\\ &\cup \{IOMatch_B : \mathbf{closr} \to B \to o \mid B \in \mathbb{B}'\},\end{aligned}$$

  although this contains top-level relational variables from Δ, which never appear in t.
- α is a valuation of s such that if $v \in \texttt{FV}(s) : \rho$, where ρ is a relational arrow sort, there exists $c \in A'_{\mathbf{closr}}$ such that $\text{expand}_\alpha(c) = \alpha(v)$.

- α' is a valuation of t satisfying
  - $\alpha'(v) = c$ for $v \in \mathtt{FV}(s) \setminus \mathtt{dom}(\Delta)$ such that $\mathrm{expand}_\alpha(c) = \alpha(v)$.
  - $\alpha'(Apply_B) = apply_B$ for $B \in \mathbb{B}'$.
  - $\alpha'(IOMatch_B) = iomatch_B$ for $B \in \mathbb{B}'$.

  Note that $X \notin \mathtt{dom}(\alpha')$. Here, $apply_B$ and $iomatch_B$ are defined in Section 4.3.2.

Notice that $apply_B$ in α' is not monotone if $B = o$ as explained in Appendix B. This will not be problematic, since I do not rely on monotonicity of α' to prove the first direction of meaning preservation.

Moreover, given a term u, if no existential quantifiers in u bind higher-order variables and all symbols in u have order at most 2, $\mathcal{M}[\![u]\!](\alpha') = \mathcal{S}[\![u]\!](\alpha')$. The monotone and standard semantics differ when we have existential quantifiers over higher-order variables. Hence, if we use α' to interpret first-order goal terms, the monotone and standard semantics give the same interpretation. This can be formally proved by induction on the grammar of goal terms. Thus, within this section, I write $\mathcal{M}[\![u]\!](\alpha')$ even though α' is not truly monotone.

In addition, $T^{\mathcal{M}}_{P':\Delta'}$ is equivalent to $T^{\mathcal{S}}_{P':\Delta'}$, although $T^{\mathcal{M}}_{P':\Delta'}$ is not guaranteed to be monotone if an input is not drawn from $\mathcal{M}[\![\Delta']\!]$. The monotonicity of $T^{\mathcal{M}}_{P':\Delta'}$ is not used in this section.

The next lemma establishes a semantic relationship between a source goal term and a target goal term.

Lemma 4.4.1. *If $\sigma \notin \mathbb{B}$, we have a unique $c \in A'_{\mathbf{closr}}$ such that $\mathcal{M}[\![\Gamma' \vdash t : \mathbf{closr}]\!](\alpha' \cup [X \mapsto c]) = 1$. In addition, this c satisfies $expand_\alpha(c) = \mathcal{M}[\![\Gamma \vdash s : \sigma]\!](\alpha)$. Otherwise, if σ is a base sort, we have $\mathcal{M}[\![\Gamma \vdash s : b]\!](\alpha) = \mathcal{M}[\![\Gamma' \vdash t : b]\!](\alpha')$.*

The uniqueness of $c \in A'_{\mathbf{closr}}$ in the above lemma is important for the inductive proof to work. It is worth noting that this c is a closure object that represents the partially applied function s.

This lemma allows us to establish the first direction of meaning preservation.

Theorem 4.4.1. *If $\mathcal{P}$ is solvable, so is $\mathcal{P}'$.*

4.4.2 Continuity of one-step consequence operators

There are difficulties with applying the idea of valuation extracting to prove the second direction of meaning preservation (see Appendix C.2). Hence, for the second direction, I adopt a different approach that does not involve valuation extraction. My approach was originally inspired by the work on Communicating Sequential Processes by Roscoe [1997], though it later turned out that in the literature on logic programming, the same approach has been used for a long time [Lloyd, 1987; Hogger, 1990].

In this section, I introduce the notion of "continuity", also known as Scott continuity.

Given a partially ordered set (poset) P and a subset $X \subseteq P$, the greatest lower bound of X is denoted by $\bigsqcap X$ and the least upper bound of X is denoted by $\bigsqcup X$.

It is explained in [Cathcart Burn et al., 2018] that $\mathcal{M}[\![\Delta]\!]$ is a complete lattice. I will first introduce several key definitions taken from [Roscoe, 1997].

Definition 4.4.1. *Given a poset P, a subset $D \subseteq P$ is said to be directed if each finite subset F of D has an upper bound in D; in other words, there is $y \in D$ such that $x \leq y$ for all $x \in F$.*

Definition 4.4.2. *A complete partial order (often abbreviated cpo) is a partial order in which every directed set has a least upper bound, and which has a least element (denoted by $\bot$).*

A complete lattice is also a complete partial order.

Definition 4.4.3. *If P and Q are two complete partial orders and $f : P \to Q$, then f is said to be continuous if, whenever $R \subseteq P$ is directed, $\bigsqcup\{f(x) \mid x \in R\}$ exists and equals $f(\bigsqcup R)$.*

A continuous function can be shown to be monotone, although I will not do it here. The next proposition establishes that $T^{\mathcal{M}}_{P:\Delta}$ is continuous when the underlying complete lattice is finite.

Proposition 4.4.1. *If $\mathcal{M}[\![\Delta]\!]$ is finite, then $T^{\mathcal{M}}_{P:\Delta} : \mathcal{M}[\![\Delta]\!] \to \mathcal{M}[\![\Delta]\!]$ is continuous.*

Continuity of $T^{\mathcal{M}}_{P:\Delta}$ is a strictly weaker condition than the finiteness of $\mathcal{M}[\![\Delta]\!]$. For instance, even when some universes A_b are infinite, if $T^{\mathcal{M}}_{P:\Delta}$ is an identity function, it is continuous.

The next theorem shows that if $T^{\mathcal{M}}_{P:\Delta}$ is continuous, then there exists a constructive way to obtain a fixed point.

Theorem 4.4.2. *If f is continuous, then $\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$ is the least fixed point of f.*

4.4.3 Second direction

The next lemma shows that this diagram commutes:

$$\begin{array}{ccc} \zeta & \xrightarrow{T_f} & \zeta' \\ {\scriptstyle T^{\mathcal{M}}_{P:\Delta}}\uparrow & & \uparrow{\scriptstyle T^{\mathcal{M}}_{P':\Delta'}} \\ \gamma & \xrightarrow[T_f]{} & \gamma' \end{array}$$

Lemma 4.4.2. *Given a valuation γ of P and a valuation γ' of P', suppose $\gamma' = T_f(\gamma)$ holds. If $\zeta = T^{\mathcal{M}}_{P:\Delta}(\gamma)$ and $\zeta' = T^{\mathcal{M}}_{P':\Delta'}(\gamma')$, then $\zeta' = T_f(\zeta)$.*

The next lemma states that T_f holds between the lowest upper bounds of two increasing sequences whose valuations are related by T_f.

Lemma 4.4.3. *Assume* $\beta = \bigsqcup\{f_1^n(\alpha) \mid n \in \mathbb{N}\}$, *where* $f_1 = T^{\mathcal{M}}_{P:\Delta}$, *and* $\beta' = \bigsqcup\{f_2^n(\alpha') \mid n \in \mathbb{N}\}$, *where* $f_2 = T^{\mathcal{M}}_{P':\Delta'}$. *If* $\alpha' = T_f(\alpha)$, *then* $\beta' = T_f(\beta)$.

The next theorem establishes soundness of the defunctionalization algorithm, albeit under the extra assumption that one-step consequence operators for the source and target problems are continuous.

Theorem 4.4.3. *Given that* $T^{\mathcal{M}}_{P:\Delta}$ *and* $T^{\mathcal{M}}_{P':\Delta'}$ *are continuous, if* $\mathcal{P}'$ *is solvable, then so is* $\mathcal{P}$.

4.4.4 Continuous semantics

Recent work by Jochems [2018] studies the continuous semantics, which uses continuous function spaces to interpret goal terms. In his working paper, it is shown that one-step consequence operators in the continuous semantics are continuous:

Theorem 4.4.4. $T^{\mathcal{C}}_{P:\Delta}$ *is continuous for all programs* P *in the continuous semantics.*

Further, Jochems [2018] proves the equivalence between the monotone and continuous semantics:

Theorem 4.4.5. *The HoCHC safety problem* (Δ, P, G) *is solvable under the monotone interpretation, if and only if it is solvable under the continuous interpretation.*

In Section 4.4.3, the key result is Lemma 4.4.2, which in turn hinges on Lemma 4.4.1. Valuation extraction works correctly even if we start with a source "continuous" problem (as opposed to a source monotone problem). Because the defunctionalized target problem is first order, it has the same meaning regardless of which of the standard, monotone, and continuous semantics we use to interpret the problem. Therefore, Lemma 4.4.1 can be adapted to the continuous semantics. Also, Lemma 4.4.2 can be adapted to the continuous semantics. As a consequence, adapting Theorem 4.4.3 to the continuous semantics yields

Theorem 4.4.6. *If* $\mathcal{P}'$ *is solvable under the continuous semantics, then* $\mathcal{P}$ *is also solvable under the continuous semantics.*

This is because one-step consequence operators in the continuous semantics are continuous by Theorem 4.4.4.

Finally, this gives

$$\begin{aligned}
\mathcal{P}' \text{ is solvable under } \mathcal{M} &\implies \mathcal{P}' \text{ is solvable under } \mathcal{C} && \text{by Theorem 4.4.5} \\
&\implies \mathcal{P} \text{ is solvable under } \mathcal{C} && \text{by Theorem 4.4.6} \\
&\implies \mathcal{P} \text{ is solvable under } \mathcal{M} && \text{by Theorem 4.4.5},
\end{aligned}$$

where $\mathcal{M}$ and $\mathcal{C}$ denote the monotone and continuous semantics, respectively. Therefore, the defunctionalization algorithm is sound.

Chapter 5

Implementation and evaluation

This document describes how the defunctionalization algorithm for monotone problems is implemented. The source code, including a test suite, is available at https://github.com/LongPham7/Defunctionalization-of-monotone-problems. A web interface is available at http://mjolnir.cs.ox.ac.uk/dfhochc/. This web interface feeds the defunctionalization algorithm's output into Z3, an SMT solver developed by Microsoft Research, to verify the defunctionalized target problems.

5.1 Implementation

5.1.1 Input format

By way of example, consider a monotone safety problem $\mathcal{P} = (\Delta, P, G)$, where

$$\begin{aligned}
\Delta &= \{add : \texttt{int} \to \texttt{int} \to \texttt{int} \to \texttt{bool}, \\
&\qquad twice : (\texttt{int} \to \texttt{int} \to \texttt{bool}) \to \texttt{int} \to \texttt{int} \to \texttt{bool}\} \\
P &= \{add = \lambda x{:}\texttt{int}, y{:}\texttt{int}, z{:}\texttt{int}.x + y = z, \\
&\qquad twice = \lambda f{:}\texttt{int} \to \texttt{int} \to \texttt{bool}, x{:}\texttt{int}, y{:}\texttt{int}.(\exists_{\texttt{int}} z.f\ x\ z \wedge f\ z\ y)\} \\
G &= \exists_{\texttt{int}} x.add\ 1\ 2\ x.
\end{aligned}$$

For simplicity, DefMono only handles the background theory of linear integer arithmetic (ZLA).

An input file corresponding to the monotone problem above is

```
# This is a sample comment.

environment
add: int -> int -> int -> bool
twice: (int -> int -> bool) -> int -> int -> bool

program
```

```
8  add := \x: int. \y: int. \z:int. x + y = z;
9  twice := \f: int -> int -> bool. \x: int. \y:int. E z:int. f x z && f z y;
10
11 goal
12 E x: int. add 1 2 x
```

As can be seen in line 1, single-line comments start with `#`. Multiline comments are not supported.

A sort environment is placed in the `environment` section. Each statement in the sort environment is allowed to span multiple lines, without endmarkers. By contrast, under the `program` section, each equation defining a top-level relational variable must end with a semicolon. This restriction is placed to make parsing easier.

The binding operator λ in a lambda calculus is written as `\`, and $\exists$ is written as `E`. The sorts of variables bound by λ and $\exists$ must be specified.

Following the notation in Haskell, conjunction is written as `&&`, and disjunction is written as `||`.

For first-order formulas from ZLA, the following operators are included: `<`, `<=`, `=`, `>`, and `>=`. Inequality such as $a \neq b$ can be expressed by `a < b || a > b`.

5.1.2 Output format

DefMono supports two output formats. One is the same format as that of inputs, which is preferable if a readable output is desired. The other format is the 'pure' SMT-LIB2 format, and it allows outputs to be readily fed into Z3.

Since target problems produced by the defunctionalization algorithm involve closures (i.e. entities of the **closr** sort), it is necessary to encode them. This is achieved by using a list-like algebraic data type with equality. The following example demonstrates how to define closures in a suitable manner for Z3. *Twice* and *Add* are top-level relational variables in this example.

```
1 (declare-datatypes () ((Closr
2   Twice
3   Add
4   (boolCons (boolHd Bool) (boolTl Closr))
5   (intCons (intHd Int) (intTl Closr))
6   (closrCons (closrHd Closr) (closrTl Closr)) )))
```

The name of the algebraic data type, `Closr`, is stated in line 1. In lines 2 and 3, `Twice` and `Add` represent $(Twice) \in A'_{\mathbf{closr}}$ and $(Add) \in A'_{\mathbf{closr}}$. In lines 4–6, `boolCons`, `intCons`, and `closrCons` are data constructors that append Booleans, integers, and closures, respectively, to input closures. Each of these constructors comes with selector functions for heads and tails of lists. Note that in the encoding of $(X, t_1, \ldots, t_k)$, where X is a top-level relational variable, the head is t_n rather than X.

5.2 Evaluation

In this section, I evaluate the performance of DefMono in respect of its verification capability and its running time. Additionally, its performance is compared with that of two other higher-order verification tools:

- HORUS[1] by Cathcart Burn et al. [2018]: this runs a refinement type-based algorithm on higher-order Horn clause problems.
- MoCHi[2] by Kobayashi et al. [2011]; Sato et al.: this runs a CEGAR-based model checking algorithm on higher-order verification problems written in OCaml.

The test suite for DefMono is obtained from that for HORUS by adding one additional test case: 'hold'. 'hold' is originally presented in Section 5.3 of [Cathcart Burn et al., 2018] as an example that is beyond HORUS's verification capability. HORUS's test suite is obtained from MoCHi's. As some of them use the list datatype, which is not supported by HORUS and DefMono, such test cases are disregarded. The remaining test cases were then translated from OCaml into Horn clause problems by Cathcart Burn et al. [2018].

5.2.1 Verification capability

The verification outcomes are summarised in Table 5.1. An input problem being solvable is indicated by `sat` in DefMono and HORUS and by `safe` in MoCHi. In fact, the output of HORUS is `unsat` when an input is solvable; however, for readability, it is reversed.

For HORUS and MoCHi, I used their web interfaces to collect the results. The Z3 used in a web server running HORUS's web interface is version 4.4.1. As for DefMono, I used Z3 version 4.6.0.

According to [Kobayashi et al., 2011], MoCHi verifies all test cases in HORUS correctly. Furthermore, because 'hold' is solvable, MoCHi verifies it correctly as well [Cathcart Burn et al., 2018]. In all test cases except 'a-max' and 'a-max-e', because the outputs of DefMono coincide with those of MoCHi, DefMono verifies these test cases correctly as well. Regarding 'a-max' and 'a-max-e', DefMono does not terminate within two minutes. This shows that DefMono's outputs may be out of Z3's reach. In this test suite, MoCHi returns unsafe if and only if DefMono terminates and returns `unsat`. Hence, none of the test cases violates completeness or soundness of DefMono.

With respect to HORUS, 'neg' and 'hold' demonstrate incompleteness of HORUS (i.e. they are solvable, but their respective transforms are not typable). Thus, DefMono is more capable than HORUS with respect to 'neg' and 'hold'. On the other hand, HORUS correctly verifies 'a-max' and 'a-max-e', which cannot be handled by DefMono.

[1] The source code of HORUS can be found at `https://github.com/penteract/HigherOrderHornRefinement`. The web interface is available at `http://mjolnir.cs.ox.ac.uk/horus/`.

[2] The web interface of MoCHi is available at `http://www-kb.is.s.u-tokyo.ac.jp/~ryosuke/mochi/`. Since the original paper [Kobayashi et al., 2011] on MoCHi was published, this web interface has incorporated an extension described in [Unno et al., 2013].

Test case	HORUS	MoCHi	DefMono
ack	sat	safe	sat
a-max	sat	safe	time out
a-max-e	sat	safe	time out
herc	sat	safe	sat
max	sat	safe	sat
mc91	sat	safe	sat
mc91-e	unsat	unsafe	unsat
mult	sat	safe	sat
mult-e	unsat	unsafe	unsat
neg	unsat	safe	sat
repeat-e	unsat	unsafe	unsat
sum	sat	safe	sat
sum-e	unsat	unsafe	unsat
hold	unsat	safe	sat

Table 5.1: Verification outcomes of HORUS, MoCHi, and DefMono

5.2.2 Running time

The running time of DefMono and HORUS on the test suite is presented in Table 5.2 and Table 5.3. The column 'Def' shows the the running time of the defunctionalization algorithm. The column 'Solving' shows the running time of Z3 v.4.6.0 to solve target monotone problems generated by the defunctionalization algorithm. The column 'Trans' shows the execution time of transforming an input higher-order Horn clause problem into a first-order one using refinement types. The experiment was conducted on Windows 10 using an Intel Core i7 CPU.

The running time of Z3 to solve target problems varies greatly from test case to test case: the execution time ranges from 16.72 ms in 'sum-e' to 3.35 s in 'max'. Moreover, as explained before, Z3 does not terminate on 'a-max' and 'a-max-e' within two minutes.

As for HORUS, all in all, it takes less time for transformation than DefMono does for defunctionalization. 'repeat-e' is the only test case where DefMono is faster than HORUS. In 'repeat-e', the difference in their running time is 5.14 ms. In the remaining test cases, the differences fall between 0.22 ms (in 'sum') and 25.15 ms (in 'a-max').

As for Z3's execution time, HORUS is mostly faster than DefMono. The only exceptions are 'mc91-e', 'sum', and 'sum-e', although the differences between HORUS and HORUS in these test cases are insignificant. Moreover, the differences between HORUS and DefMono in Z3's execution time are considerable in some cases. For instance, in 'max', it takes 3.35 s for Z3 to solve the target monotone problem generated by DefMono, whereas it only takes 20.80 ms in HORUS—several orders of magnitude smaller.

Test case	Def (ms)	Solving (ms)
ack	25.15	34.48
a-max	41.51	time out
a-max-e	40.36	time out
hrec	26.58	76.05
max	37.92	3347.60
mc91	21.39	35.66
mc91-e	23.91	20.14
mult	22.06	58.19
mult-e	14.10	70.38
neg	16.76	367.21
repeat-e	16.32	361.41
sum	13.94	23.75
sum-e	14.05	16.72
hold	13.84	31.83

Table 5.2: Running time of DefMono

Test case	Trans (ms)	Solving (ms)
ack	14.37	24.03
a-max	16.36	36.60
a-max-e	16.54	38.33
hrec	15.77	34.04
max	15.27	20.80
mc91	13.45	27.15
mc91-e	18.94	22.80
mult	13.41	30.41
mult-e	13.43	24.39
neg	15.91	24.47
repeat-e	21.46	22.63
sum	13.72	25.49
sum-e	13.81	20.71
hold	13.39	18.02

Table 5.3: Running time of HORUS

Chapter 6

Conclusion

6.1 Conclusion

Reynolds's defunctionalization is a viable approach to reducing HoCHC to first-order constrained Horn clauses. In this paper, I have presented an algorithm to defunctionalize HoCHC into first-order constrained Horn clause problems. Additionally, I have proved the following:

1. Type preservation: outputs of the algorithm are well-sorted.
2. Completeness: if a source HoCHC problem is solvable, the target first-order constrained Horn clause problem generated by the defunctionalization algorithm is also solvable.
3. Soundness: if the target problem is solvable, the source problem is also solvable.

Therefore, type preservation and meaning preservation (i.e. completeness and soundness) have been established in this work.

In addition to the theoretical work, I have implemented a system named DefMono that uses the defunctionalization algorithm to verify programs. I have also compared DefMono's performance with that of other higher-order verification tools, HORUS and MoCHi. In respect of verification capability, DefMono is less capable than MoCHi because Z3 cannot solve defunctionalized problems of some test cases within two minutes. In comparison with HORUS, DefMono can correctly verify some test cases that HORUS cannot handle. However, HORUS does not present any bottleneck in Z3's processing of HORUS's outputs, whilst DefMono can cause Z3 to time out. With respect to running time, the defunctionalization-based approach is slower than HORUS. This is probably because target problems produced by DefMono use a more complicated background theory than the background theory of source problems.

6.2 Future work

I propose three continuations of the present work.

Continuity of one-step consequence operators Whether one-step consequence operators in the monotone semantics are continuous is an interesting question in its own right. I attempted to prove continuity of one-step consequence operators by structural induction on goal terms, as done in the proof of their monotonicity. However, I encountered a difficulty in the inductive case of function applications: the least upper bound operator $\bigsqcup$ is not guaranteed to distribute over function applications. Hence, I believe this is a key to finding a counterexample. In fact, a counterexample to continuity of monotone one-step consequence operators has been found and is presented in a working paper by Jerome Jochems at the University of Oxford. This counterexample shows that the least upper bound operator does not always distribute over function applications.

Theory of closures One weakness of the defunctionalization-based reduction of higher-order Horn clause problems to first-order ones is that the background theories of target problems involve closures. In DefMono, closures are implemented using an algebraic data type. Fortunately, algebraic data types can be handled by Z3, thanks to recent advances in Horn-clause solving technology. Without these advances, it would have been impossible to verify target problems produced by the defunctionalization algorithm. Hence, it is another avenue of future work to study, for instance, how ZLA coupled closures can be more efficiently handled in Horn-clause solving.

Implementation One direction is to extend the test suite. As of now, all test cases have order at most 2. Hence, it will be interesting to investigate how DefMono handles test cases of higher order.

Another direction is to investigate why DefMono does not seem to terminate on 'a-max' and 'a-max-e'. The run time statistics of Z3 show that only one Boolean variable is created when 'a-max' is tested. This is extremely odd because in other cases where Z3 terminates, many Boolean variables are created. It is therefore likely that Z3 never halts on 'a-max'.

Appendix A

Supplements for the defunctionalization algorithm

A.1 Preprocessing

Let the source monotone problem be $\mathcal{P} = (\Delta, P, G)$. Prior to defunctionalization $\mathcal{P}$, we need to eliminate all anonymous functions in P and G and then perform η-expansion to fully expand the outermost lambda abstractions defining top-level relational variables.

Every equation in P can be expressed as

$$X{:}\sigma_1 \to \cdots \to \sigma_m \to o = \lambda x_1, \ldots, x_n.E, \tag{A.1}$$

where $m \leq n$ and E is not a lambda abstraction.

Anonymous functions refer to lambda abstractions occurring inside E in (A.1). Suppose that E contains the anonymous function

$$\Delta \vdash \lambda x{:}\sigma.F : \sigma \to \rho.$$

Further, assume that the set of free variables occurring in $\lambda x.F$ is

$$\texttt{FV}(\lambda x.F) = \{n_1, \ldots, n_k\}$$

and that $\Delta \vdash n_i : \sigma_i$ for all $1 \leq i \leq k$. The definition of a fresh top-level relational variable X' is then added to P:

$$X' = \lambda n_1{:}\sigma_1, \ldots, n_k{:}\sigma_k, x{:}\sigma.F. \tag{A.2}$$

As the the actual parameters for the free variables $\{n_1, \ldots, n_k\}$ are specified outside $\lambda x.F$, we need to use lambda abstraction to pass these parameters. The anonymous function $\lambda x.F$ is then replaced with

$$X'\ n_1\ \cdots\ n_k.$$

This process of moving local functions (that is, anonymous functions) into a global scope is called *lambda lifting* in the literature.

We repeat the same step for all the remaining anonymous functions in P and G. Notice that some anonymous functions may be inside the definition of X'. In order to use a fresh top-level relational variable for each step, the anonymous functions are eliminated one by one sequentially rather than concurrently. This procedure terminates because the number of anonymous functions is finite.

Once all anonymous functions are turned into equations, η-expansion is performed on the right hand side of every equation from P. This is guided by the following inference rules:

$$\frac{E \rightsquigarrow_\eta F}{\lambda x.E \rightsquigarrow_\eta \lambda x.F} \qquad \frac{\Delta \vdash E : \sigma_1 \to \cdots \to \sigma_m \to o \qquad E \neq \lambda x.F \text{ for any } F}{E \rightsquigarrow_\eta (\lambda x_1{:}\sigma_1, \ldots, x_m{:}\sigma_m.E\ x_1\ \cdots\ x_m)}$$

The result of η-expansion on F is obtained by applying $\rightsquigarrow_\eta$ on F. The inference rule on the right encompasses the case when $m = 0$. In that case, we have $\lambda\overline{x}.E \rightsquigarrow_\eta \lambda\overline{x}.E$, where $E : o$. This transformation is applied to the right hand side of every equation in P.

A.2 Rationale for the algorithm design

When a source term is of the form $E\ F$, either (App) or (Match) is applied, depending on whether the function application returns a term of an arrow sort or of base sort. One of its premises of (App) is $(E\ F) \rightsquigarrow_A^X H$, where $\rightsquigarrow_A^X$ is defined by (App-Base) and (App-Arrow). Which of these two rules is applied is determined by whether F has a base sort. In both (App-Base) and (App-Arrow), neither premises nor conclusions use $\rightsquigarrow_A^X$. Thus, we could remove $\rightsquigarrow_A^X$ completely from the inference rules by merging (App) with each of (App-Base) and (App-Arrow). The reason why I do not do this is that the resulting inference rules would be too long to fit the width of a page. This is why $\rightsquigarrow_A^X$ and $\rightsquigarrow_M$ are necessary.

It is worth observing that $\rightsquigarrow$ is only applicable when the source term is of base sort and $\rightsquigarrow^X$ is only applicable when the source term has an arrow sort. This is a rule I imposed on the inference rules to reduce their complexity.

To explain my reasoning, consider the target term of $Add\ x$ in (4.5). Applying the identity $y = C^0_{Add}$, we can write the target term more succinctly as

$$Apply_{\mathbf{int}}\ C^0_{Add}\ x\ X.$$

In order to have the inference rules produce this succinct form, we need to split the rule (App-Base) into two rules corresponding to two cases: the case when E' is a logical formula and the case when E' is a single variable symbol. In the first case, we cannot write $Apply\ E'\ F'\ X$, since E' is a logical formula rather than a variable symbol. By contrast, in the second case, $Apply\ E'\ F'\ X$ is a valid target term.

If this idea were implemented, we would have $Add \rightsquigarrow C^0_{Add}$ instead of $Add \rightsquigarrow^X X = C^0_{Add}$. The former is more natural and less confusing than the latter. However, it does

not seem elegant to split (App-Base), because we would need to work out whether E' consists only of a single symbol. Also, splitting (App-Base) will increase the total number of inference rules. Therefore, I opted to enforce the rule that whenever the source term has an arrow sort, the parameter X can be passed. Consequently, when a source goal term is a top-level relational variable, the term has an arrow sort and hence its target term must accept a parameter. This is the reason behind the bizarre looking (TopVar).

Appendix B

Monotonicity of extracted valuations

This chapter presents how to establish monotonicity of α', which is formally defined in Section 4.3.2.

B.1 Preliminaries

First, I prove a lemma that characterizes orders of higher-order elements.

Lemma B.1.1. *Assume that f_1 and f_2 have sort $\sigma_1 \to \cdots \to \sigma_k \to o$, where $k \geq 0$ and each σ_i is either a relational arrow sort or a base sort. Then $f_1 \subseteq f_2$ if and only if for each $t \in \mathcal{M}[\![\sigma_1]\!] \times \cdots \times \mathcal{M}[\![\sigma_k]\!]$, we have $f_1(t) \subseteq_o f_2(t)$.*

Proof. For both directions, the claim is proved by induction on k. In this proof, I use curried notation and non-curried notation interchangeably. Hence, if an n-tuple is input to a function, the n components of the tuple are fed into the function separately.

First, I prove ($\Rightarrow$). For the base case, when $k = 0$, we have $f_1, f_2 : o$. By assumption, $f_1 \subseteq_o f_2$ and hence the claim holds.

For the inductive case, suppose that $f_1 \subseteq f_2$. By the definition of $\subseteq$, for all $c_1 \in \mathcal{M}[\![\sigma_1]\!]$, we have $f_1\ c_1 \subseteq f_2\ c_1$. This is true regardless of whether σ_1 is a relational sort or a non-propositional base sort. Now by the inductive hypothesis, as $f_1\ c_1 \subseteq f_2\ c_1$, for all $(c_2, \ldots, c_k) \in \mathcal{M}[\![\sigma_2]\!] \times \cdots \times \mathcal{M}[\![\sigma_k]\!]$, we obtain

$$(f_1\ c_1)(c_2, \ldots, c_k) \subseteq_o (f_2\ c_1)(c_2, \ldots, c_k).$$

Thus, for all $c_1, \ldots, c_k$ of appropriate sorts,

$$f_1(c_1, \ldots, c_k) \subseteq_o f_2(c_1, \ldots, c_k),$$

as required.

Now I turn to ($\Leftarrow$). For the base case, when $k = 0$, if $f_1 \subseteq_o f_2$, the claim immediately follows.

For the inductive case, suppose that for all for all $(c_1, \ldots, c_k) \in \mathcal{M}[\![\sigma_1]\!] \times \cdots \times \mathcal{M}[\![\sigma_k]\!]$, we have

$$f_1(c_1, \ldots, c_k) \subseteq_o f_2(c_1, \ldots, c_k).$$

Fix arbitrary $c_1 \in \mathcal{M}[\![\sigma_1]\!]$. Then for all $(c_2, \ldots, c_k) \in \mathcal{M}[\![\sigma_2]\!] \times \cdots \times \mathcal{M}[\![\sigma_k]\!]$, we have

$$(f_1\ c_1)(c_2, \ldots, c_k) \subseteq_o (f_2\ c_1)(c_2, \ldots, c_k).$$

Hence, by the inductive hypothesis, $f_1\ c_1 \subseteq f_2\ c_1$. Because c_1 is arbitrary, by the definition of $\subseteq$, $f_1 \subseteq f_2$. This concludes the proof. □

The next lemma characterizes monotone functions.

Lemma B.1.2. *Assume $f \in \mathcal{M}[\![\sigma_1]\!] \Rightarrow \cdots \Rightarrow \mathcal{M}[\![\sigma_k]\!] \Rightarrow \mathbb{2}$, where $k \geq 0$ and each σ_i is either a relational arrow sort or a base sort. f is monotone if and only if for each $t_1, t_2 \in \mathcal{M}[\![\sigma_1]\!] \times \cdots \times \mathcal{M}[\![\sigma_k]\!]$ and $t_1 \subseteq t_2$, we have $f(t_1) \subseteq_o f(t_2)$. Here, $t_1 \subseteq t_2$ holds if and only if the order holds in each component.*

Proof. For both directions, the claim is proved by induction on k. In this proof, I use curried notation and non-curried notation interchangeably. Hence, if an n-tuple is input to a function, the n components of the tuple are fed into the function separately.

I first start with ($\Rightarrow$). For the base case, when $k = 0$, the claim is clearly true.

For the inductive case, suppose that f is monotone. By definition, we have

$$\mathcal{M}[\![\sigma_1 \to \cdots \to \sigma_k \to o]\!] = \mathcal{M}[\![\sigma_1]\!] \Rightarrow_m \mathcal{M}[\![\sigma_2 \to \cdots \to \sigma_k \to o]\!].$$

It follows from the definition of $\Rightarrow_m$ that for any $c_1, d_1 \in \mathcal{M}[\![\sigma_1]\!]$, if $c_1 \subseteq d_1$, then $X\ c_1 \subseteq X\ d_1$. Thus, it follows from Lemma B.1.1 that

$$(f\ c_1)(c_2, \ldots, c_k) \subseteq_o (f\ d_1)(c_2, \ldots, c_k). \tag{B.1}$$

Furthermore, because $f\ d_1$ is monotone, by the inductive hypothesis, for any $(c_2, \ldots, c_k)$ and $(d_2, \ldots, d_k)$ from $\mathcal{M}[\![\sigma_2]\!] \times \cdots \times \mathcal{M}[\![\sigma_k]\!]$ such that $(c_2, \ldots, c_k) \subseteq (d_2, \ldots, d_k)$, we have

$$(f\ d_1)(c_2, \ldots, c_k) \subseteq_o (f\ d_1)(d_2, \ldots, d_k). \tag{B.2}$$

Combining (B.1) and (B.2) gives

$$(f\ c_1)(c_2, \ldots, c_k) \subseteq_o (f\ d_1)(d_2, \ldots, d_k).$$

Therefore, for any $t_1, t_2 \in \mathcal{M}[\![\sigma_1]\!] \times \cdots \times \mathcal{M}[\![\sigma_k]\!]$ such that $t_1 \subseteq t_2$, we have

$$f(t_1) \subseteq_o f(t_2),$$

as required.

Now I turn to ($\Leftarrow$). For the base case, when $k = 0$, the claim is vacuously true.

For the inductive case, by assumption, for any $t_1, t_2 \in \mathcal{M}[\![\sigma_1]\!] \times \cdots \times \mathcal{M}[\![\sigma_k]\!]$ such that $t_1 \subseteq t_2$, we have $f(t_1) \subseteq_o f(t_2)$. Now fix $c_1, d_1 \in \mathcal{M}[\![\sigma_1]\!]$ such that $c_1 \subseteq d_1$. Then by the assumption, for any $(c_2, \ldots, c_k), (d_2, \ldots, d_k) \in \mathcal{M}[\![\sigma_2]\!] \times \cdots \times \mathcal{M}[\![\sigma_k]\!]$ such that $(c_2, \ldots, c_k) \subseteq (d_2, \ldots, d_k)$, we have

$$(f\ c_1)(c_2, \ldots, c_k) \subseteq_o (f\ c_1)(d_2, \ldots, d_k) \tag{B.3}$$
$$(f\ d_1)(c_2, \ldots, c_k) \subseteq_o (f\ d_1)(d_2, \ldots, d_k) \tag{B.4}$$
$$(f\ c_1)(c_2, \ldots, c_k) \subseteq_o (f\ d_1)(c_2, \ldots, c_k). \tag{B.5}$$

Applying the inductive hypothesis to (B.3) yields that $f\ c_1$ is monotone. Likewise, by the application of the inductive hypothesis to (B.4), $X\ d_1$ is also monotone. Further, from Lemma B.1.1 and (B.5), we obtain

$$f\ c_1 \subseteq f\ d_1.$$

To summarise, $f\ c_1$ and $f\ d_1$ are both monotone, and $f\ c_1 \subseteq f\ d_1$ whenever $c_1 \subseteq d_1$. Therefore, f is monotone by definition. This concludes the proof. □

B.2 Monotonicity of α'

Thus, in order for $\alpha'(X)$ to be monotone, where $\Delta' \vdash X : \sigma_1 \to \cdots \to \sigma_m \to o$, for any $t_1, t_2 \in \mathcal{M}[\![\sigma_1]\!] \times \cdots \times \mathcal{M}[\![\sigma_m]\!]$ such that $t_1 \subseteq t_2$, we should have

$$\alpha'(X)(t_1) \subseteq_o \alpha'(X)(t_2).$$

This holds for $X = IOMatch_B$, where $B \in \mathbb{B}'$. If $B \neq o$ and $t_1, t_2 \in \mathcal{M}[\![\mathbf{closr} \to B]\!]$, then $t_1 \subseteq t_2$ implies $t_1 = t_2$. Otherwise, if $B = o$, by the monotonicity of α, $iomatch_B$ is monotone as well.

However, this does not hold for $X = Apply_o$. For instance, suppose that $Y \in \mathtt{dom}(\Delta)$ and that $\Delta \vdash Y : \mathbf{nat} \to o \to \mathbf{nat} \to o$. Then,

$$((Y, 2), 0, (Y, 2, 0)) \subseteq ((Y, 2), 1, (Y, 2, 0))$$

and yet

$$apply_o\ (Y, 2)\ 0\ (Y, 2, 0) \not\subseteq_o apply_o\ (Y, 2)\ 1\ (Y, 2, 0)$$

as the left hand side evaluates to 1, whereas the right hand side evaluates to 0. Therefore, α' is not monotone.

In this way, α' is "nearly" monotone, apart from $apply_o$. $apply_B$ augments an input to an input closure, thereby simulating function application that still yields a strictly partially applied function. $\mathcal{P}$'s rough equivalent of $apply_B$ is function application. However, monotonicity of function application in $\mathcal{P}$ does not carry over to $\mathcal{P}'$, for the way $apply_B$ simulates function application is different from genuine function application in $\mathcal{P}$.

By contrast, there is a nice correspondence between branches of $IOMatch_B$ and top-level relational variables from Δ. The monotonicity of $\alpha(X)$, where $X \in \mathtt{dom}(\Delta)$, carries over to $iomatch_B$ that corresponds to X, although this is true only when $B = o$; if $B \neq o$, $iomatch_B$ is monotone regardless of monotonicity of $\alpha(X)$.

To fix the issue of monotonicity of $apply_o$, observe that α' can be interpreted as a standard valuation. This can be established by the next proposition.

Proposition B.2.1. *Assume $f : b_1 \to \cdots \to b_m$, where each b_i is a base sort. If $f \in \mathcal{M}[\![b_1]\!] \Rightarrow \cdots \Rightarrow \mathcal{M}[\![b_m]\!]$, where f is not necessarily monotone, then $f \in \mathcal{S}[\![b_1 \to \cdots \to b_m]\!]$.*

Proof. Immediately follows from the fact that $\mathcal{M}[\![b]\!] = \mathcal{S}[\![b]\!]$ if b is a base sort. □

Despite its triviality, this proposition is important. For example, consider $X : (o \to o) \to o$, which has order 3. Also, let β a "nearly" monotone valuation for X in the sense that

$$\begin{aligned}\beta(X) &\in \mathcal{M}[\![o \to o]\!] \Rightarrow \mathcal{M}[\![o]\!] \\ &= (\mathbb{2} \Rightarrow_m \mathbb{2}) \Rightarrow \mathbb{2},\end{aligned}$$

where the second $\Rightarrow$ on the second line is not $\Rightarrow_m$. When we want to extend the monotone interpretation of X to the standard semantics, there is no straightforward way to do so, since β does not define the result of $\alpha(X)$ applied to f when $f \in (\mathbb{2} \Rightarrow \mathbb{2}) \setminus (\mathbb{2} \Rightarrow_m \mathbb{2})$.

In contrast, if the sort of X has order 2, we can extend the monotone interpretation of X to the standard semantics in a straightforward fashion.

Proposition B.2.1 can be applied to any function occurring in P' because any $f \in \mathtt{dom}(\Delta')$ has order 2 (by convention, it is assumed that all top-level relational variables have arrow sorts) and any $f \in \mathtt{dom}(\mathbb{S})$ has order at most 2. Furthermore, we do not have existential quantifiers over higher-order variables in P'. For these two reasons, the standard semantics of P' coincides with the monotone semantics of P'. That is, $\mathcal{M}[\![s]\!](\alpha') = \mathcal{S}[\![s]\!](\alpha')$ holds, given that s contains no existential quantifiers over higher-order variables and all symbols occurring in s have order at most 2. Therefore, α' can be viewed as a standard valuation of P'.

In addition, $T^{\mathcal{M}}_{P':\Delta'}$ is equivalent to $T^{\mathcal{S}}_{P':\Delta'}$, although $T^{\mathcal{M}}_{P':\Delta'}$ is not guaranteed to be monotone if an input is not drawn from $\mathcal{M}[\![\Delta']\!]$. Hence, Lemma 2 in [Cathcart Burn et al., 2018] does not apply if a valuation is nearly but not monotone:

Lemma B.2.1. *$\mathcal{M}[\![\Delta \vdash G : \rho]\!] \in \mathcal{M}[\![\Delta]\!] \Rightarrow_m \mathcal{M}[\![\rho]\!]$, where G is any goal term. Also, $T^{\mathcal{M}}_{P:\Delta} \in \mathcal{M}[\![\Delta]\!] \Rightarrow_m \mathcal{M}[\![\Delta]\!]$.*

Proof. "Immediately follows from the fact that `mexists`, `and` and `or` are monotone and all the construction [in the inductive definition of $\mathcal{M}[\![\Delta \vdash G : \rho]\!]$] are monotone combinations". [Cathcart Burn et al., 2018] Note that the interpretations of constant symbols from $\mathbb{S}$ are required to be monotone as well. □

Now suppose that α' is a prefix of $T^{\mathcal{M}}_{P':\Delta'} = T^{\mathcal{S}}_{P':\Delta'}$ and satisfies $\mathcal{M}[\![G']\!](\alpha') = \mathcal{S}[\![G']\!](\alpha') = 0$, where G' is the goal formula component of $\mathcal{P}'$. In other words, suppose that α' is a solution to $\mathcal{P}'$ under the standard semantics. I restate Theorem 2.3.1 with a slightly different notation (this is originally Theorem 2 in [Cathcart Burn et al., 2018]):

Theorem B.2.1. *The higher-order constrained Horn clause problem (Δ', D', G') is solvable if and only if the monotone problem $(\Delta', P_{D'}, G')$ is solvable.*

For each monotone problem (Δ', P', G'), there exists a higher-order constrained Horn clause problem (Δ', D', G') such that $P' = P_{D'}$. Further, Horn clause problems are interpreted using the standard semantics. Consequently, we obtain Lemma 1 from [Cathcart Burn et al., 2018]:

Theorem B.2.2. *For definite formula D', the prefixed points of $T^{\mathcal{S}}_{P_{D'}}$ are exactly the models of D'.*

Finally, the next theorem ensures the existence of a monotone solution to $\mathcal{P}'$, provided that α' is a solution to $\mathcal{P}'$ under the standard semantics.

Theorem B.2.3. *If α' is a solution to $\mathcal{P}'$ under the standard semantics, then $\mathcal{P}'$ is solvable under the monotone semantics.*

Proof. Let α' be a standard solution to $\mathcal{P}'$. Suppose (Δ', D', G') is the higher-order constrained Horn clause problem that is equivalent to $\mathcal{P}'$; i.e. $P' = P_{D'}$. Such a Horn clause problem is well-defined as there is one-one correspondence between higher-order constrained Horn clause problems and monotone problems.

Since it is given that α' is a solution to $\mathcal{P}'$ and hence is a prefixed point of $T^{\mathcal{S}}_{P':\Delta'}$, by Theorem B.2.2, α' is also a model of D'. Further, $\mathcal{S}[\![G']\!](\alpha') = 0$. Hence, α' is a solution to (Δ', D', G').

Lastly, it follows from Theorem B.2.1 that $\mathcal{P}' = (\Delta', P', G')$ is solvable under the monotone semantics because $P' = P_{D'}$ by assumption. □

Appendix C

Supplements for meaning preservation

This chapter presents the proofs for the results introduced in Section 4.4 and explains difficulties with applying valuation extraction to the proof of soundness.

C.1 First direction

Lemma 4.4.1. *If $\sigma \notin \mathbb{B}$, we have a unique $c \in A'_{\mathbf{closr}}$ such that $\mathcal{M}[\![\Gamma' \vdash t : \mathbf{closr}]\!](\alpha' \cup [X \mapsto c]) = 1$. In addition, this c satisfies $expand_\alpha(c) = \mathcal{M}[\![\Gamma \vdash s : \sigma]\!](\alpha)$. Otherwise, if σ is a base sort, we have $\mathcal{M}[\![\Gamma \vdash s : b]\!](\alpha) = \mathcal{M}[\![\Gamma' \vdash t : b]\!](\alpha')$.*

Proof. The proof proceeds by structural induction on s. First, I consider the case when σ is an arrow sort.

For the base case, if $s = x$, where x is an ordinary variable of an arrow sort, t is equal to $X = x$ due to (Var-Arrow). Because x is a free variable, it must be included in $\mathtt{dom}(\alpha)$. Therefore, $c = \alpha'(x) \in A'_{\mathbf{closr}}$ works. This c is unique because if $c_1 \neq_{\mathbf{closr}} c$, then $c = \alpha(x)$ and $c_1 = \alpha(x)$ cannot hold simultaneously (this is due to the fact that $(=)$ is the same as $(=_{\mathbf{closr}})$ in this setting).

By the definition of α', we have $\mathrm{expand}_\alpha(\alpha'(x)) = \alpha(x)$. It follows that

$$\begin{aligned}\mathrm{expand}_\alpha(c) &= \mathrm{expand}_\alpha(\alpha'(x)) \\ &= \alpha(x) \\ &= \mathcal{M}[\![\Gamma \vdash x : \sigma]\!](\alpha) \\ &= \mathcal{M}[\![\Gamma \vdash s : \sigma]\!](\alpha).\end{aligned}$$

Therefore, $\mathrm{expand}_\alpha(c) = \mathcal{M}[\![\Gamma \vdash s : \sigma]\!](\alpha)$ holds.

Another base case is when $s \in \Delta$; i.e. s is a top-level relational variable. By (Top-Var), t is equal to $X = C^0_s$. The only value of c that satisfies $\mathcal{M}[\![X = C^0_s]\!](\alpha' \cup [X \mapsto c]) = 1$ is $(s) \in A'_{\mathbf{closr}}$ because C^0_s, which is a constant symbol, is by default interpreted as (s). Here, $(s) \in A'_{\mathbf{closr}}$ is a 1-tuple containing s. Thus, such c is unique. Further, we have $\mathrm{expand}_\alpha(c) = \alpha(s)$. Therefore, the claim holds.

For the inductive case, s is transformed into t by either (App-Base) or (App-Arrow).

Assume $s = E\ F$, where E has an arrow sort and F has a base sort. By (App-Base), t is equal to

$$\exists_{\mathbf{closr}} x.(E' \wedge \mathit{Apply}_{\mathbf{closr}}\ x\ F'\ X)),$$

where $E \rightsquigarrow^{x} E'$ and $F \rightsquigarrow F'$. Applying the inductive hypothesis to E, we have a unique $c_1 \in A'_{\mathbf{closr}}$ such that

$$\mathcal{M}[\![\Gamma' \vdash E' : \mathbf{closr}]\!](\alpha' \cup [x \mapsto c_1]) = 1.$$

Additionally, this c_1 satisfies

$$\mathrm{expand}_{\alpha}(c_1) = \mathcal{M}[\![E]\!](\alpha).$$

Further, applying the inductive hypothesis to F, we have $\mathcal{M}[\![F]\!](\alpha) = \mathcal{M}[\![F']\!](\alpha')$.

Consequently, we obtain

$$\begin{aligned}
& \mathcal{M}[\![\exists_{\mathbf{closr}} x.(E' \wedge \mathit{Apply}_{\mathbf{closr}}\ x\ F'\ X))]\!](\alpha' \cup [X \mapsto c]) = 1 \\
\iff & \mathcal{M}[\![\mathit{Apply}_{\mathbf{closr}}\ x\ F'\ X]\!](\alpha' \cup [x \mapsto c_1, X \mapsto c]) = 1 \\
\iff & \mathit{apply}_{\mathbf{closr}}\ c_1\ \mathcal{M}[\![F']\!](\alpha')\ c = 1 \\
\iff & (c == \mathit{append}(c_1, \mathcal{M}[\![F']\!](\alpha'))) = 1 \\
\iff & c = \mathit{append}(c_1, \mathcal{M}[\![F']\!](\alpha'))
\end{aligned}$$

Here, $(==)$ is a comparator. The second line follows from the uniqueness of c_1. The third line follows from the fact that α' interprets $\mathit{Apply}_{\mathbf{closr}}$ as $\mathit{apply}_{\mathbf{closr}}$. The fourth line follows form the definition of $\mathit{apply}_{\mathbf{closr}}$.

Thus, to satisfy $\mathcal{M}[\![\mathit{Apply}_{\mathbf{closr}}\ x\ F'\ X]\!](\alpha' \cup [x \mapsto c_1, X \mapsto c]) = 1$, we should set c to $\mathrm{append}(c_1, \mathcal{M}[\![F']\!](\alpha'))$. Hence, there is indeed a unique $c \in A'_{\mathbf{closr}}$ that satisfies $\mathcal{M}[\![\Gamma' \vdash t : \mathbf{closr}]\!](\alpha' \cup [X \mapsto c]) = 1$.

Furthermore, from $c = \mathrm{append}(c_1, \mathcal{M}[\![F']\!](\alpha'))$, we derive

$$\begin{aligned}
\mathrm{expand}_{\alpha}(c) &= \mathrm{expand}_{\alpha}(\mathrm{append}(c_1, \mathcal{M}[\![F']\!](\alpha'))) \\
&= \mathrm{expand}_{\alpha}(c_1)\ \mathrm{expand}_{\alpha}(\mathcal{M}[\![F']\!](\alpha')) \\
&= \mathcal{M}[\![E]\!](\alpha)\ \mathcal{M}[\![F']\!](\alpha')) \\
&= \mathcal{M}[\![E]\!](\alpha)\ \mathcal{M}[\![F]\!](\alpha)) \\
&= \mathcal{M}[\![E\ F]\!](\alpha).
\end{aligned}$$

The second equality follows from the inductive definition of the expand_{α} function.

Lastly, if (App-Arrow) is used, we have $s = E\ F$, and t is equal to

$$\exists_{\mathbf{closr}} x.(E' \wedge \exists_{\mathbf{closr}} y.(F' \wedge \mathit{Apply}_{\mathbf{closr}}\ x\ y\ X)),$$

where $E \rightsquigarrow^{x} E'$ and $F \rightsquigarrow^{y} F'$. By the inductive hypothesis, we have unique $c_1, c_2 \in A'_{\mathbf{closr}}$ such that

$$\begin{aligned}
&\mathcal{M}[\![\Gamma' \vdash E' : \mathbf{closr}]\!](\alpha' \cup [x \mapsto c_1]) = 1 \\
&\mathcal{M}[\![\Gamma' \vdash F' : \mathbf{closr}]\!](\alpha' \cup [y \mapsto c_2]) = 1.
\end{aligned}$$

Additionally, c_1 and c_2 satisfy

$$\begin{aligned}\text{expand}_\alpha(c_1) &= \mathcal{M}[\![E]\!](\alpha)\\ \text{expand}_\alpha(c_2) &= \mathcal{M}[\![F]\!](\alpha).\end{aligned}$$

As a consequence, we have

$$\begin{aligned}&\mathcal{M}[\![\exists_{\mathbf{closr}} x.(E' \wedge \exists_{\mathbf{closr}} y.(F' \wedge \mathit{Apply}_{\mathbf{closr}}\ x\ y\ X))]\!](\alpha' \cup [X \mapsto c]) = 1\\ \iff\ &\mathcal{M}[\![\mathit{Apply}_{\mathbf{closr}}\ x\ y\ X]\!]([x \mapsto c_1, y \mapsto c_2, X \mapsto c]) = 1\\ \iff\ &\mathit{apply}_{\mathbf{closr}}\ c_1\ c_2\ c = 1\\ \iff\ &(c == \text{append}(c_1, c_2)) = 1\\ \iff\ &c = \text{append}(c_1, c_2).\end{aligned}$$

The second line follows from the uniqueness of c_1 and c_2. The third line follows from the the interpretation of $\mathit{Apply}_{\mathbf{closr}}$ by α'. The fourth line follows from the definition of $\mathit{apply}_{\mathbf{closr}}$.

Therefore, the only value of c that satisfies $\mathcal{M}[\![\Gamma' \vdash t : \mathbf{closr}]\!](\alpha' \cup [X \mapsto c]) = 1$ is $\text{append}(c_1, c_2)$.

Moreover, $c = \text{append}(c_1, c_2)$ yields

$$\begin{aligned}\text{expand}_\alpha(\text{append}(c_1, c_2)) &= \text{expand}_\alpha(c_1)\ \text{expand}_\alpha(c_2)\\ &= \mathcal{M}[\![E]\!](\alpha)\ \mathcal{M}[\![F]\!](\alpha)\\ &= \mathcal{M}[\![E\ F]\!](\alpha).\end{aligned}$$

The first equality follows from the inductive definition of the expand_α function. Therefore, the claim is true.

Next, consider the case when σ is a base sort.

For the base case, if $s \in Fm \cup Tm$, we have $s \rightsquigarrow s$ by (ConstrLan). All free variables occurring in first-order terms from a constraint language have base sorts (this is proved in Theorem D.1.1). Further, since $\text{expand}_\alpha(c) = c$ when c is of base sort from $\mathbb{B}$, α and α' have the same interpretation of all free variables in s. Also, A and A' have the same universes for each $b \in \mathbb{B}$ and have the same interpretation of constant symbols from $\mathbb{S}$. Therefore, s has the same meaning in both $\langle A, \alpha\rangle$ and $\langle A', \alpha'\rangle$. Thus, the claim is true.

The case when (Var-Base) is used can be proved straightforwardly.

For the inductive case, if $s = E \wedge F$, by (LogSym), t is equal to $E' \wedge F'$, where $E \rightsquigarrow E'$ and $F \rightsquigarrow F'$. It follows from the inductive hypothesis that

$$\begin{aligned}\mathcal{M}[\![E]\!](\alpha) &= \mathcal{M}[\![E']\!](\alpha')\\ \mathcal{M}[\![F]\!](\alpha) &= \mathcal{M}[\![F']\!](\alpha').\end{aligned}$$

Therefore, we obtain

$$\begin{aligned}\mathcal{M}[\![E' \wedge F']\!](\alpha') &= \mathcal{M}[\![E']\!](\alpha') \wedge \mathcal{M}[\![F']\!](\alpha')\\ &= \mathcal{M}[\![E]\!](\alpha) \wedge \mathcal{M}[\![F]\!](\alpha)\\ &= \mathcal{M}[\![E \wedge F]\!](\alpha)\end{aligned}$$

as required. The case for s being $E \vee F$ can be proved in the same manner.

If $s = \exists_b x.F$, by (Exi), we have $t = \exists_b x.F'$, where $F \rightsquigarrow F'$. By the inductive hypothesis,

$$\mathcal{M}[\![F]\!](\alpha \cup [x \mapsto c]) = \mathcal{M}[\![F']\!](\alpha' \cup [x \mapsto c])$$

for any $c \in A_b = A'_b$. Thus, we obtain

$$\begin{aligned}\mathcal{M}[\![\exists_b x.F']\!](\alpha') &= \exists c \in A'_b.\mathcal{M}[\![F']\!](\alpha' \cup [x \mapsto c]) \\ &= \exists c \in A_b.\mathcal{M}[\![F]\!](\alpha \cup [x \mapsto c]) \\ &= \mathcal{M}[\![\exists_b x.F]\!](\alpha).\end{aligned}$$

Therefore, the claim holds.

It is essential that the existential quantifier is bound to a variable of base sort as opposed to an arrow sort. If $\exists_\sigma x.F \rightsquigarrow \exists_{\mathbf{closr}} x.F'$, where σ is an arrow sort, it is possible that $\mathcal{M}[\![\exists_\sigma x.F]\!](\alpha) = 1$ and yet $\mathcal{M}[\![\exists_{\mathbf{closr}} x.F']\!](\alpha') = 0$. This is because $\mathcal{M}[\![\sigma]\!]$ contains functions that cannot be represented by any element of $A'_{\mathbf{closr}}$. This is why we need to eliminate existential quantifiers over higher-order variables.

Next, assume $s = E\ F$. If F is of base sort, (Match-Base) is applied to defunctionalize s into t, yielding

$$t = \exists_{\mathbf{closr}} x.(E' \wedge IOMatch_\sigma\ x\ F'),$$

where $E \rightsquigarrow^x E'$ and $F \rightsquigarrow F'$. Applying the inductive hypothesis to E, we have a unique $c \in A'_{\mathbf{closr}}$ such that

$$\mathcal{M}[\![\Gamma' \vdash E' : \mathbf{closr}]\!](\alpha' \cup [x \mapsto c]) = 1.$$

Also, this c satisfies $\mathrm{expand}_\alpha(c) = \mathcal{M}[\![E]\!](\alpha)$. Additionally, applying the inductive hypothesis to F, we have $\mathcal{M}[\![F]\!](\alpha) = \mathcal{M}[\![F']\!](\alpha')$.

By the uniqueness of c,

$$\mathcal{M}[\![\exists_{\mathbf{closr}} x.(E' \wedge IOMatch_\sigma\ x\ F')]\!](\alpha') = \mathcal{M}[\![IOMatch_\sigma\ x\ F')]\!](\alpha' \cup [x \mapsto c]).$$

Furthermore, we obtain

$$\begin{aligned}\mathcal{M}[\![IOMatch_\sigma\ x\ F')]\!](\alpha' \cup [x \mapsto c]) &= iomatch_\sigma\ c\ \mathcal{M}[\![F']\!](\alpha') \\ &= iomatch_\sigma\ c\ \mathcal{M}[\![F]\!](\alpha) \\ &= \mathrm{expand}_\alpha(c)\ \mathrm{expand}_\alpha(\mathcal{M}[\![F]\!](\alpha)) \\ &= \mathcal{M}[\![E]\!](\alpha)\ \mathcal{M}[\![F]\!](\alpha) \\ &= \mathcal{M}[\![E\ F]\!](\alpha).\end{aligned}$$

The second equality follows from the identity $\mathcal{M}[\![F]\!](\alpha) = \mathcal{M}[\![F']\!](\alpha')$. The third equality follows from the definition of $iomatch_\sigma$. The fourth equality follows from the definition of c and the definition of expand_α when the input has a base sort.

Therefore, $\mathcal{M}[\![\Gamma \vdash s : b]\!](\alpha) = \mathcal{M}[\![\Gamma' \vdash t : b]\!](\alpha')$ holds.

Otherwise, if F is of an arrow sort, (Match-Arrow) is applied to defunctionalize s into t, where t is

$$\exists_{\mathbf{closr}} x.(E' \wedge \exists_{\mathbf{closr}} y.IOMatch_{\mathbf{closr}}\ x\ y),$$

where $E \rightsquigarrow^{x} E'$ and $F \rightsquigarrow^{y} F'$. By the inductive hypothesis, we have unique $c_1, c_2 \in A'_{\mathbf{closr}}$ such that

$$\mathcal{M}[\![\Gamma' \vdash E' : \mathbf{closr}]\!](\alpha' \cup [x \mapsto c_1]) = 1$$
$$\mathcal{M}[\![\Gamma' \vdash F' : \mathbf{closr}]\!](\alpha' \cup [y \mapsto c_2]) = 1.$$

Further, c_1 and c_2 satisfy

$$\mathrm{expand}_{\alpha}(c_1) = \mathcal{M}[\![E]\!](\alpha)$$
$$\mathrm{expand}_{\alpha}(c_2) = \mathcal{M}[\![F]\!](\alpha).$$

By the uniqueness of c_1 and c_2,

$$\begin{aligned}\mathcal{M}[\![\exists_{\mathbf{closr}} x.(E' \wedge \exists_{\mathbf{closr}} y.IOMatch_{\mathbf{closr}}\ x\ y)]\!](\alpha') &= \mathcal{M}[\![IOMatch_{\mathbf{closr}}\ x\ y)]\!](\alpha' \cup [x \mapsto c_1, y \mapsto c_2]) \\ &= \mathcal{M}[\![IOMatch_{\mathbf{closr}}\ x\ y)]\!]([x \mapsto c_1, y \mapsto c_2]).\end{aligned}$$

The only free variables in $IOMatch_{\mathbf{closr}}\ x\ y$ are x and y. Hence, α' does not affect its semantics; thus, the second equality follows. The above expression can be further reduced to

$$\begin{aligned}\mathcal{M}[\![IOMatch_{\mathbf{closr}}\ x\ y)]\!]([x \mapsto c_1, y \mapsto c_2]) &= iomatch_{\mathbf{closr}}\ c_1\ c_2 \\ &= \mathrm{expand}_{\alpha}(c_1)\ \mathrm{expand}_{\alpha}(c_2) \\ &= \mathcal{M}[\![E]\!](\alpha)\ \mathcal{M}[\![F]\!](\alpha) \\ &= \mathcal{M}[\![E\ F]\!](\alpha).\end{aligned}$$

Therefore, the claim holds. This concludes the proof. □

Theorem C.1.1. *If α is a model of P, then $\alpha' = T_f(\alpha)$ is a model for P'.*

Proof. Assume that α is a model of P; that is, α is a prefixed point of $T^{\mathcal{M}}_{P:\Delta}$. It is given that

$$\mathcal{M}[\![\Delta \vdash P(X) : \Delta(X)]\!](\alpha) \subseteq_{\Delta(X)} \alpha(X) \tag{C.1}$$

for each $X \in \mathtt{dom}(\Delta)$. In addition, since $\alpha \in \mathcal{M}[\![\Delta]\!]$, we have $\mathtt{dom}(\alpha) = \mathtt{dom}(\Delta)$. Suppose $P(X)$ is of the form

$$\lambda x_1, \ldots, x_m.F,$$

where $\Delta \vdash X : \sigma_1 \to \cdots \to \sigma_m \to o$. X gives rise to

$$IOMatch_{\sigma'_m} = \lambda x, x_m.(\exists x_1, \ldots, x_{m-1}.x = C_X^{m-1}\ x_1\ \cdots\ x_{m-1} \wedge F'),$$

where $\sigma_m \rightsquigarrow_T \sigma'_m$ and $F \rightsquigarrow F'$. The sort of $IOMatch_{\sigma'_m}$ is $\mathbf{closr} \to \sigma'_m \to o$.

Now suppose that for some $c \in A'_{\mathbf{closr}}$ and $c_m \in A'_{\sigma'_m}$, we have

$$\mathcal{M}[\![\exists x_1, \ldots, x_{m-1}.x = C_X^{m-1}\ x_1\ \cdots\ x_{m-1} \wedge F']\!](\alpha' \cup [x \mapsto c, x_m \mapsto c_m]) = 1.$$

This means there exists $c_i \in A'_{\sigma'_i}$ for each $1 \leq i < m$ such that

$$c = (X, c_1, \ldots, c_{m-1}) \tag{C.2}$$
$$\mathcal{M}[\![F']\!](\alpha' \cup \{[x_i \mapsto c_i] \mid 1 \leq i \leq m\}) = 1. \tag{C.3}$$

Let β' be the valuation $\{(x_i, c_i) \mid 1 \leq i \leq m\}$. For simplicity, I write $\{(x_i, c_i) \mid 1 \leq i \leq m\}$ for $\{[x_i \mapsto c_i] \mid 1 \leq i \leq m\}$. Also, let β be $\{(x_i, \mathrm{expand}_\alpha(c_i)) \mid 1 \leq i \leq m\}$. Because $\mathtt{FV}(F) \subseteq \mathtt{dom}(\alpha) \cup \mathtt{dom}(\beta)$, $\alpha \cup \beta$ is a valid valuation for F. Similarly, $\alpha' \cup \beta'$ is a valid valuation of F'.

By Lemma 4.4.1 and (C.3),

$$\begin{aligned}\mathcal{M}[\![F]\!](\alpha \cup \beta) &= \mathcal{M}[\![F']\!](\alpha' \cup \beta') \\ &= 1.\end{aligned}$$

Because $\mathcal{M}[\![F]\!](\alpha \cup \beta) = 1$ and $P(X) = F$, it follows from (C.1) that

$$\mathcal{M}[\![X\ x_1\ \cdots\ x_m]\!](\alpha \cup \beta) = 1$$

Therefore, we obtain

$$\begin{aligned}\mathcal{M}[\![IOMatch_{\sigma'_m}\ x\ x_m]\!](\alpha' \cup \beta') &= iomatch_{\sigma'_m}\ c\ c_m \\ &= \mathrm{expand}_\alpha(c)\ \mathrm{expand}_\alpha(c_m) \\ &= \mathrm{expand}_\alpha((X, c_1, \ldots, c_{m-1}))\ \mathrm{expand}_\alpha(c_m) \\ &= \alpha(X)\ \mathrm{expand}_\alpha(c_1)\ \cdots\ \mathrm{expand}_\alpha(c_m) \\ &= \mathcal{M}[\![X\ x_1\ \cdots\ x_m]\!](\alpha \cup \beta) \\ &= 1.\end{aligned}$$

Thus, for all $c \in A'_{\mathbf{closr}}$ and $c_m \in A'_{\sigma'_m}$,

$$\begin{aligned}&\mathcal{M}[\![\exists x_1, \ldots, x_{m-1}.x = C_X^{m-1}\ x_1\ \cdots\ x_{m-1} \wedge F']\!](\alpha' \cup [x \mapsto c, x_m \mapsto c_m]) \\ \subseteq_o\ &\mathcal{M}[\![IOMatch_{\sigma'_m}\ x\ x_m]\!](\alpha' \cup [x \mapsto c, x_m \mapsto c_m]).\end{aligned} \tag{C.4}$$

Hence, we obtain

$$\begin{aligned}&\mathcal{M}[\![\lambda x, x_m.\exists x_1, \ldots, x_{m-1}.x = C_X^{m-1}\ x_1\ \cdots\ x_{m-1} \wedge F']\!](\alpha') \\ \subseteq_{\mathbf{closr} \to \sigma'_m \to o}\ &\mathcal{M}[\![IOMatch_{\sigma'_m}]\!](\alpha').\end{aligned}$$

Even if $IOMatch_B$ has multiple branches corresponding to different top-level relational variables from Δ, the disjunction of the left hand side of (C.4) for each X contributing to

$IOMatch_B$ is smaller than or equal to the right hand side of (C.4). It therefore follows that

$$\mathcal{M}[\![P'(IOMatch_B)]\!](\alpha') \subseteq_{\mathbf{closr}\to B\to o} \alpha'(IOMatch_B).$$

If $X = Apply_B$, by the definition of $apply_B$,

$$\mathcal{M}[\![P'(Apply_B)]\!](\alpha') = \alpha'(Apply_B)$$
$$\therefore \mathcal{M}[\![P'(Apply_B)]\!](\alpha') \subseteq_{\mathbf{closr}\to B\to\mathbf{closr}\to o} \alpha'(Apply_B).$$

Hence, for every $X \in \mathtt{dom}(\Delta')$,

$$\mathcal{M}[\![P'(X)]\!](\alpha') \subseteq_{\Delta'(X)} \alpha'(X).$$

As α' is a prefixed point of $T^{\mathcal{M}}_{P':\Delta'}$ (which is equivalent to $T^{\mathcal{S}}_{P':\Delta'}$), it is indeed a model of P'. This concludes the proof. □

Theorem 4.4.1. *If $\mathcal{P}$ is solvable, so is $\mathcal{P}'$.*

Proof. Let α be a solution to $\mathcal{P}$ and α' be a valuation for P' derived from α. By Theorem C.1.1, α' is a model of P'. Furthermore, since $G \rightsquigarrow G'$, it follows from Lemma 4.4.1 that

$$\mathcal{M}[\![G]\!](\alpha) = \mathcal{M}[\![G']\!](\alpha').$$

Because α is a solution to $\mathcal{P}'$, $\mathcal{M}[\![G]\!](\alpha) = 0$. Therefore, $\mathcal{M}[\![G']\!](\alpha') = 0$ as well. Hence, α' is a solution to $\mathcal{P}'$ under the monotone semantics. □

C.2 Difficulties with valuation extraction in the second direction

For the second direction of meaning preservation, I explain some difficulties in extracting solutions to $\mathcal{P}$ from solutions to $\mathcal{P}'$ as we did for the first direction. Consider the example introduced in Section 4.3.1. Suppose that a solution to $\mathcal{P}'$ is

$$\alpha' = \{IOMatch_{\mathbf{nat}} \mapsto iomatch_{\mathbf{nat}}, Apply_{\mathbf{nat}} \mapsto apply_{\mathbf{nat}}, Apply_{\mathbf{closr}} \mapsto apply_{\mathbf{closr}}\},$$

where $apply_{\mathbf{nat}}$ and $apply_{\mathbf{closr}}$ are defined (independently of α) in Section 4.3.1. The interpretation of $IOMatch_{\mathbf{nat}}$ is

$$iomatch_{\mathbf{nat}} = add' \cup twice',$$

where the functions $add' : A'_{\mathbf{closr}} \to \mathbb{N} \to \mathbb{2}$ and $twice' : A'_{\mathbf{closr}} \to \mathbb{N} \to \mathbb{2}$ are

$$add'\ m\ n = \begin{cases} 1 & \text{if } m = (Add, n_1, n_2), n = n_1 + n_2 \\ 0 & \text{otherwise} \end{cases}$$

and

$$twice'\ m\ n = \begin{cases} 1 & \text{if } m = (Twice, f, n_1) \\ & \quad \wedge \exists n_2.((\exists n_3.apply_{\mathbf{nat}}\ f\ n_1\ n_3 \wedge iomatch_{\mathbf{nat}}\ n_3\ n_2) \\ & \quad \wedge (\exists n_4.apply_{\mathbf{nat}}\ f\ n_2\ n_4 \wedge iomatch_{\mathbf{nat}}\ n_4\ n)) \\ 0 & \text{otherwise.} \end{cases}$$

There are three issues with extracting a valuation for P from α'.

1. It is not straightforward to define a valuation for P that has the same structure as α. For instance, because the sort of Add in $\mathcal{P}'$ has order 2 (i.e. not a higher-order function), add' can be straightforwardly transferred to the interpretation of Add, yielding $add : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to o$ given as

$$add\ n_1\ n_2\ n_3 = \begin{cases} 1 & \text{if } n_3 = n_1 + n_2 \\ 0 & \text{otherwise.} \end{cases}$$

   On the other hand, it is not easy to extract an interpretation for $Twice$ from $twice'$. This is because $twice'$ is defined in terms of $iomatch_{\mathbf{nat}}$, which is in turn defined in terms of $twice'$ (and add').

   Due to this recursive nature of the definition of $twice'$, it is not clear how to construct a valuation α for P that satisfies $\alpha' = T_f(\alpha)$. It is crucial for α to have the same structure as α' because it lets us apply Lemma 4.4.1 to prove $\mathcal{M}[\![G]\!](\alpha) = 0$.

2. Suppose that the first issue is overcome and that α that satisfies $\alpha' = T_f(\alpha)$ has been obtained. With the same example as above, it is natural to have

$$\alpha(Twice)\ f\ n_1\ n_2 = 0$$

   whenever f is not expressible in $\mathcal{P}$; that is, whenever f cannot be expressed by combination of $Twice$ and Add. This creates an issue that α is not monotone. For example,

$$(\lambda a, b.a + 1 = b) \subseteq_\rho U_\rho$$

   but

$$\alpha(Twice)\ (\lambda a, b.a + 1 = b)\ 2\ 4 \not\subseteq_o \alpha(Twice)\ U_\rho\ 2\ 4. \tag{C.5}$$

   As $\lambda a, b.a + 1 = b$ can be expressed by $Add\ 1$, the left hand side of (C.5) evaluates to 1. However, since U_ρ cannot be expressed by any element of $A'_{\mathbf{closr}}$, the right hand side of (C.5) evaluates to 0. Thus, $\alpha(Twice)$ is not monotone; hence, neither is α.

3. The third problem with α is that it is not necessarily a prefixed point of $T^{\mathcal{M}}_{P:\Delta}$. In the above example,

$$\mathcal{M}[\![\lambda f, a, b.(\exists c.f\ a\ c \wedge f\ c\ b)]\!](\alpha) \not\subseteq_{\Delta(Twice)} \alpha(Twice)$$

   holds since the left hand side can take $f = U_\rho$ and produces 1 for any a and b, whilst the right hand side does not.

C.3 Continuity of one-step consequence operators

Proposition 4.4.1. *If $\mathcal{M}[\![\Delta]\!]$ is finite, then $T^{\mathcal{M}}_{P:\Delta} : \mathcal{M}[\![\Delta]\!] \to \mathcal{M}[\![\Delta]\!]$ is continuous.*

Proof. I will prove that for every directed subset $R \subseteq \mathcal{M}[\![\Delta]\!]$, $\bigsqcup\{T^{\mathcal{M}}_{P:\Delta}(x) \mid x \in R\}$ exists and equals $T^{\mathcal{M}}_{P:\Delta}(\bigsqcup R)$.

Fix $R \subseteq \mathcal{M}[\![\Delta]\!]$. Because $T^{\mathcal{M}}_{P:\Delta}$ is monotone, it is given

$$\forall x \in R.x \subseteq \bigsqcup R$$
$$\therefore \forall x \in R.T^{\mathcal{M}}_{P:\Delta}(x) \subseteq T^{\mathcal{M}}_{P:\Delta}(\bigsqcup R).$$

Note that the order of valuations is denoted by $\subseteq$ rather than $\leq$. Thus,

$$\bigsqcup\{T^{\mathcal{M}}_{P:\Delta}(x) \mid x \in R\} \subseteq T^{\mathcal{M}}_{P:\Delta}(\bigsqcup R), \tag{C.6}$$

where the left hand side exists as $\mathcal{M}[\![\Delta]\!]$ is a complete lattice.

It remains to prove that both sides of the above inequality are in fact equal. If $\mathcal{M}[\![\Delta]\!]$ is finite, then R must be finite as well. Since R is directed by assumption and is finite, $\bigsqcup R \in R$. Hence,

$$T^{\mathcal{M}}_{P:\Delta}(\bigsqcup R) \subseteq \bigsqcup\{T^{\mathcal{M}}_{P:\Delta}(x) \mid x \in R\}. \tag{C.7}$$

Combining (C.6) and (C.7), we obtain

$$\bigsqcup\{T^{\mathcal{M}}_{P:\Delta}(x) \mid x \in R\} = T^{\mathcal{M}}_{P:\Delta}(\bigsqcup R).$$

Therefore, $T^{\mathcal{M}}_{P:\Delta}$ is indeed continuous if $\mathcal{M}[\![\Delta]\!]$ is finite. □

Theorem 4.4.2. *If f is continuous, then $\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$ is the least fixed point of f.*

Proof. Since f is continuous, it is monotone. Therefore, $\langle f^n(\bot) \mid n \in \mathbb{N}\rangle$ is an increasing sequence. As f is continuous,

$$f(\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}) = \bigsqcup\{f^{n+1}(\bot) \mid n \in \mathbb{N}\}.$$

Because $\bot$ cannot be larger than any element from $\{f^{n+1}(\bot) \mid n \in \mathbb{N}\}$,

$$\bigsqcup\{f^{n+1}(\bot) \mid n \in \mathbb{N}\} = \bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}.$$

Combining the above two equations gives

$$f(\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}) = \bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}.$$

Therefore, $\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$ is a fixed point of f.

If y is also a least fixed point of f, we have $\bot \leq y$. The monotonicity of f gives that

$$\forall n \in \mathbb{N}.f^n(\bot) \leq y,$$

which means that y is also an upper bound of $\{f^n(\bot) \mid n \in \mathbb{N}\}$. As $\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$ is the least upper bound of $\{f^n(\bot) \mid n \in \mathbb{N}\}$ by definition, $\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\} \leq y$. Thus, $\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$ is the least fixed point of f. □

C.4 Second direction

Lemma 4.4.2. *Given a valuation γ of P and a valuation γ' of P', suppose $\gamma' = T_f(\gamma)$ holds. If $\zeta = T^{\mathcal{M}}_{P:\Delta}(\gamma)$ and $\zeta' = T^{\mathcal{M}}_{P':\Delta'}(\gamma')$, then $\zeta' = T_f(\zeta)$.*

Proof. The claim can be depicted by the following commutative diagram:

$$\begin{array}{ccc} \zeta & \xrightarrow{T_f} & \zeta' \\ {\scriptstyle T^{\mathcal{M}}_{P:\Delta}}\uparrow & & \uparrow{\scriptstyle T^{\mathcal{M}}_{P':\Delta'}} \\ \gamma & \xrightarrow[T_f]{} & \gamma' \end{array}$$

Fix $X \in \mathtt{dom}(\Delta)$ and assume $\Delta \vdash X : \sigma_1 \to \cdots \to \sigma_m \to o$. Also, suppose that P contains $X = \lambda x_1, \ldots, x_m.F$. In addition, for each $1 \leq i \leq m$, consider $c_i \in \mathcal{M}[\![\sigma_i]\!]$ such that there exists $c'_i \in A'_{\sigma'_i}$ that satisfies $\mathrm{expand}_\gamma(c'_i) = c_i$. Here, $\sigma_i \rightsquigarrow_T \sigma'_i$ for each i.

Now let η be the valuation $\{(x_i, c_i) \mid 1 \leq i \leq m\}$ and η' be $\{(x_i, c'_i) \mid 1 \leq i \leq m\}$. Because all free variables, including top-level relational variables, in F are in the domains of γ and η, $\gamma \cup \eta$ is a valid valuation of F. Similarly, $\gamma' \cup \eta'$ is a valid valuation of F', where $F \rightsquigarrow F'$. Since F does not contain lambda abstractions, we can apply Lemma 4.4.1 to obtain

$$\mathcal{M}[\![F]\!](\gamma \cup \eta) = \mathcal{M}[\![F']\!](\gamma' \cup \eta'). \tag{C.8}$$

The left hand side of (C.8) is equal to

$$\begin{aligned} \mathcal{M}[\![F]\!](\gamma \cup \eta) &= \mathcal{M}[\![\lambda x_1, \ldots, x_m.F]\!](\gamma)\ c_1\ \cdots\ c_m \\ &= \mathcal{M}[\![P(X)]\!](\gamma)\ c_1\ \cdots\ c_m \\ &= T^{\mathcal{M}}_{P:\Delta}(\gamma)(X)\ c_1\ \cdots\ c_m \\ &= \zeta(X)\ c_1\ \cdots\ c_m, \end{aligned} \tag{C.9}$$

where the third equality follows from the definition of $T^{\mathcal{M}}_{P:\Delta}$ and the last equality follows from the definition of ζ. The right hand side of (C.8) can be transformed into

$$\begin{aligned} &\mathcal{M}[\![F']\!](\gamma' \cup \eta') \\ &= \mathcal{M}[\![\lambda x, x_m.\exists x_1, \ldots, x_{m-1}.x = C^{m-1}_X\ x_1\ \cdots\ x_{m-1} \wedge F']\!](\gamma')\ (X, c'_1, \ldots, c'_{m-1})\ c'_m \\ &= \mathcal{M}[\![P'(IOMatch_{\sigma'_m})]\!](\gamma')\ (X, c'_1, \ldots, c'_{m-1})\ c'_m \\ &= T^{\mathcal{M}}_{P':\Delta'}(\gamma')(IOMatch_{\sigma'_m})\ (X, c'_1, \ldots, c'_{m-1})\ c'_m \\ &= \zeta'(IOMatch_{\sigma'_m})\ (X, c'_1, \ldots, c'_{m-1})\ c'_m. \end{aligned} \tag{C.10}$$

Combining (C.8), (C.9), and (C.10), we obtain

$$\zeta(X)\ c_1\ \cdots\ c_m = \zeta'(IOMatch_{\sigma'_m})\ (X, c'_1, \ldots, c'_{m-1})\ c'_m.$$

Therefore, it follows from the definition of T_f that $\zeta' = T_f(\zeta)$. This concludes the proof. □

Lemma 4.4.3. *Assume* $\beta = \bigsqcup\{f_1^n(\alpha) \mid n \in \mathbb{N}\}$, *where* $f_1 = T^{\mathcal{M}}_{P:\Delta}$, *and* $\beta' = \bigsqcup\{f_2^n(\alpha') \mid n \in \mathbb{N}\}$, *where* $f_2 = T^{\mathcal{M}}_{P':\Delta'}$. *If* $\alpha' = T_f(\alpha)$, *then* $\beta' = T_f(\beta)$.

Proof. As usual, fix $X \in \mathtt{dom}(\Delta)$ and assume $\Delta \vdash X : \sigma_1 \to \cdots \to \sigma_m \to o$. Also, suppose that P contains $X = \lambda x_1, \ldots, x_m.F$. In addition, for each $1 \leq i \leq m$, consider $c_i \in \mathcal{M}[\![\sigma_i]\!]$ such that there exists $c_i' \in A'_{\sigma_i'}$ that satisfies $\mathrm{expand}_\alpha(c_i') = c_i$. Here, $\sigma_i \rightsquigarrow_T \sigma_i'$ for each i.

Since β and β' are the least upper bounds of $\{f_1^n(\alpha) \mid n \in \mathbb{N}\}$ and $\{f_2^n(\alpha') \mid n \in \mathbb{N}\}$, respectively, it is given that

$$\beta(X)\ c_1\ \cdots\ c_m = \max\{f_1^n(\alpha)(X)\ c_1\ \cdots\ c_m \mid n \in \mathbb{N}\} \tag{C.11}$$

and

$$\begin{aligned} &\beta'(IOMatch_{\sigma_m'})\ (X, c_1', \ldots, c_{m-1}')\ c_m' \\ =\ &\max\{f_2^n(\alpha')(IOMatch_{\sigma_m'})\ (X, c_1', \ldots, c_{m-1}')\ c_m' \mid n \in \mathbb{N}\}. \end{aligned} \tag{C.12}$$

As $\alpha' = T_f(\alpha)$, by Lemma 4.4.2, $f_2^n(\alpha') = T_f(f_1^n(\alpha))$ for every $n \in \mathbb{N}$. Hence,

$$f_1^n(\alpha)(X)\ c_1\ \cdots\ c_m = f_2^n(\alpha')(IOMatch_{\sigma_m'})\ (X, c_1', \ldots, c_{m-1}')\ c_m'$$

for each n. Consequently,

$$\max\{f_1^n(\alpha)(X)\ c_1\ \cdots\ c_m\} = \max\{f_2^n(\alpha')(IOMatch_{\sigma_m'})\ (X, c_1', \ldots, c_{m-1}')\ c_m'\}. \tag{C.13}$$

Combining (C.11), (C.12), and (C.13) yields

$$\beta(X)\ c_1\ \cdots\ c_m = \beta'(IOMatch_{\sigma_m'})\ (X, c_1', \ldots, c_{m-1}')\ c_m'.$$

Therefore, $\beta' = T_f(\beta)$ indeed holds. □

Theorem 4.4.3. *Given that* $T^{\mathcal{M}}_{P:\Delta}$ *and* $T^{\mathcal{M}}_{P':\Delta'}$ *are continuous, if* $\mathcal{P}'$ *is solvable, then so is* $\mathcal{P}$.

Proof. Let $\bot$ be the least element from $\mathcal{M}[\![\Delta]\!]$ and $\bot'$ be the least element from $\mathcal{M}[\![\Delta']\!]$. It is clear that $\bot' = T_f(\bot)$.

Suppose $\beta = \bigsqcup\{f_1^n(\bot) \mid n \in \mathbb{N}\}$, where $f_1 = T^{\mathcal{M}}_{P:\Delta}$, and $\beta' = \bigsqcup\{f_2^n(\bot') \mid n \in \mathbb{N}\}$, where $f_2 = T^{\mathcal{M}}_{P':\Delta'}$. Because it is given that $T^{\mathcal{M}}_{P:\Delta}$ and $T^{\mathcal{M}}_{P':\Delta'}$ are continuous, β and β' are fixed points of the respective one-step consequence operators. In other words, they are models of P and P', respectively.

Furthermore, because β is obtained by iteratively applying $T^{\mathcal{M}}_{P':\Delta'}$ to $\bot'$, which is the least element of $\mathcal{M}[\![\Delta']\!]$, β' is the least fixed point of $T^{\mathcal{M}}_{P':\Delta}$ by Theorem 4.4.2. Moreover, it is the least prefixed point. This statement is not too difficult to prove, although I will not provide its formal proof.

Assume that $\mathcal{P}'$ is solvable and let its solution be α'. Then $\beta' \subseteq \alpha'$ because β' is the least model of P. Moreover, by the monotonicity of $\mathcal{M}[\![G]\!]$, we should have

$$\mathcal{M}[\![G]\!](\beta') \subseteq_o \mathcal{M}[\![G]\!](\alpha').$$

The right hand side of this equation is 0 since α' is a solution to $\mathcal{P}'$. Thus, $\mathcal{M}[\![G]\!](\beta') = 0$.

It follows from Lemma 4.4.3 that $\beta' = T_f(\beta)$. Hence, by Lemma 4.4.1, we have

$$\begin{aligned}\mathcal{M}[\![G]\!](\beta) &= \mathcal{M}[\![G]\!](\beta') \\ &= 0.\end{aligned}$$

Therefore, β is a solution to $\mathcal{P}$. This concludes the proof. □

Appendix D

Type preservation

Before I prove type preservation, I revisit the basic concepts of first-order terms from constraint languages and goal terms.

D.1 Defining terms and formulas

In this section, I formally define first-order terms and first-order formulas in constraint languages. This is necessary because to prove type preservation, I need to use some properties of terms.

D.1.1 Terms and formulas

Let $\Sigma = (\mathbb{B}, \mathbb{S})$ be a first-order signature. Since Σ is first-order, the sort of each symbol in $\mathbb{S}$ has order at most 2. The class of well-sorted first-order terms over Σ is given by

$$(\textsc{TCst})\ \frac{}{\Delta \vdash c : \mathbb{S}(c)}\ c \in \mathtt{dom}(\mathbb{S}) \qquad (\textsc{TAnd})\ \frac{}{\Delta \vdash \wedge : o \to o \to o}$$

$$(\textsc{TNeg})\ \frac{}{\Delta \vdash \neg : o \to o} \qquad (\textsc{TVar})\ \frac{}{\Delta_1, x : b, \Delta_2 \vdash x : b}$$

$$(\textsc{TExi})\ \frac{\Delta, x : b \vdash t : o}{\Delta \vdash \exists_b x.t : o} \qquad (\textsc{TApp})\ \frac{\Delta \vdash t_1 : b \to \beta \qquad \Delta \vdash t_2 : b}{\Delta \vdash t_1\ t_2 : \beta}$$

Here, b is a base sort t (with or without subscripts) is a first-order term, and β is a sort of order at most 2; i.e. sort of the form $b_1 \to \cdots \to b_n$, where $b_i \in \mathbb{B}$ for each $1 \leq i \leq n$.

It is important that Δ contains no conflicts; i.e. no variable is associated with multiple types. Henceforth, it is implicitly assumed that sort environments for first-order terms are free of conflicts.

Well-sorted first-order formulas are defined as well-sorted first-order terms of sort o. Notice that unlike in usual presentation of first-order logic, where formulas and terms are disjoint, according to the above definition, terms include formulas.

When a first-order term s is well-sorted under sort environment Δ and has sort β, I write $\Delta \vdash s : \beta$. From now on, I assume that first-order terms are well-sorted.

When a typing judgement $\Delta \vdash s : \beta$ is created by (TCst), (TAnd), or (TNeg), the sort of s is independent of Δ. In that case, to work out the sort of s, we need to check $\mathbb{S}$ and `LSym`. When $\mathbb{S}$ is unclear, I write $\mathbb{S}, \Delta \vdash s : \beta$ to make $\mathbb{S}$ explicit. However, whenever $\mathbb{S}$ is clear from the context, I will omit it from sort environments.

D.1.2 Properties of terms and formulas

Proposition D.1.1. *Every free variable occurring in a first-order term has a base sort.*

Proof. Variables can only be introduced into first-order terms by (TVar). The rule requires variables to be of base sort. Hence, the claim is true. □

Proposition D.1.2. *Given $\Delta \vdash s : \beta$, the sort of s under Δ is unique; that is, we cannot have $\Delta \vdash s : \beta'$, where $\beta \neq \beta'$.*

Proof. The proof goes by structural induction on s.

For the base case, if $\Delta \vdash s : \beta$ is created by (TCst), (TAnd), or (TNeg), the sort of s is unique (and is independent of Δ). If $\Delta \vdash s : \beta$ is created by (TVar), the sort of s is uniquely determined by Δ.

For the inductive case, if $\Delta \vdash s : \beta$ is created by (TExi), we have $\beta = o$. Thus, the sort of s is uniquely determined.

Finally, if $\Delta \vdash s : \beta$ is generated by (TApp), we know that $s = t_1\ t_2$. By the inductive hypothesis, the sorts of t_1 and t_2 under Δ are uniquely determined. Therefore, the sort of s under Δ is also uniquely determined. □

The following proposition states that each well-sorted first-order term has a unique way to assign sorts to all symbols occurring in the term such that the term is well-sorted.

Theorem D.1.1. *If $\Delta \vdash s : \beta$ holds, where s is a first-order term, then every symbol occurring in s can be annotated with a unique sort.*

Proof. The claim is proved by structural induction on s.

For the base case, if $\Delta \vdash s : \beta$ is created by (TCst), (TAnd), or (TNeg), the sort of s is given by $\mathbb{S}$ or `LSym` and is unique. If s is created by (TVar), the sort of s is given by $\Delta(s)$ and is unique because Δ is assumed to contain no conflicts. Thus, in all base cases, the sort of s can be uniquely identified. Alternatively, we can use Proposition D.1.2 to prove the base case. As the only symbol appearing in s is itself, the claim reduces to Proposition D.1.2.

For the inductive case, if s is created by (TExi), s is in the form of $\exists_b x.t$, where $\Delta, x : b \vdash t : o$. The sort of x is stored in the subscript of $\exists_b$ in s. Hence, from $\Delta \vdash \exists_b x.t : o$, we can uniquely derive $\Delta, x : b \vdash t : o$. In other words, from a conclusion of (TExi), we can uniquely deduce the corresponding premise of (TExi). By the inductive hypothesis, every symbol in $\Delta, x : b \vdash t : o$ can be annotated with a unique sort. If there exist two distinct ways to assign sorts to the symbols occurring in $\Delta \vdash \exists_b x.t : o$, there should be two distinct ways to assign sorts to $\Delta, x : b \vdash t : o$ as well, which contradicts

the inductive hypothesis. Hence, all symbols in $\exists_b x.t$ can be annotated with a unique sort.

Finally, if s is created by (TApp), we have $s = t_1\ t_2$, where $\Delta \vdash t_1 : b \to \beta$ and $\Delta \vdash t_2 : b$. We cannot determine the typing judgements $\Delta \vdash t_1 : b \to \beta$ and $\Delta \vdash t_2 : b$ uniquely by the mere appearance of $\Delta \vdash t_1\ t_2 : \beta$, without any calculation. However, we can evaluate the sorts of t_1 and t_2 under the sort environment Δ by repeatedly applying the six typing rules listed above. Furthermore, by Proposition D.1.2, the sorts of t_1 and t_2 under Δ are unique. By the inductive hypothesis, every symbol in t_1 and t_2 can be annotated with a unique symbol. For the sake of contradiction, assume that there are two distinct ways to assign sorts to $t_1\ t_2$. Then at least one of $\Delta \vdash t_1$ and $\Delta \vdash t_2$ has two distinct sort assignments. This contradicts the inductive hypothesis. Therefore, the claim holds for $t_1\ t_2$ as well. This concludes the proof. □

In effect, Theorem D.1.1 proves uniqueness of typing derivation trees of first-order terms by showing that given the root of a derivation tree, the root's successor(s) can be uniquely determined. Because all constants and variables appear at the leaves of a tree, their sort assignment is uniquely determined. As for logical constants, their sorts are given by `LSym` and hence are unique.

The syntax and typing rules of first-order terms allow us to determine the sort of each symbol in a term by simply consulting $\mathbb{S}$, `LSym`, and Δ. This nice property does not hold any longer if we omit subscripts from $\exists$. For instance, consider $\vdash (\exists x.x = 2) : o$. It is still possible to uniquely determine the sort of each symbol. However, we cannot apply the same proof as the one for Theorem D.1.1, since it is not straightforward to deduce the typing judgement $x : \texttt{int} \vdash (x = 2) : o$ (especially the left hand side of the judgement; i.e. $x : \texttt{int}$) from $\vdash (\exists x.x = 2) : o$. To determine the sort of x, we need to carry out type inference using $\vdash (=) : \texttt{int} \to \texttt{int} \to o$.

D.2 Redefining goal terms

In this section, I redefine goal terms in order to fix my imprecise use of terminology. In my explanation of the defunctionalization algorithm (Section 4.2), I call an input of transformation a 'source goal term' and an output a 'target goal term'. A problem lies in the use of the word 'goal term'. According to [Cathcart Burn et al., 2018], elements of Tm, where Tm is a set of first-order terms in a constraint language, do not qualify as goal terms. However, in my presentation of the defunctionalization algorithm, a 'source goal term' can be an element from Tm. This issue is caused by the fact that although $t \in Tm$ can be a subexpression of a goal term, t itself is not a goal term. Hence, I need to find a suitable word to refer to a collection of both goal terms and terms from Tm. One solution I would suggest is to redefine goal terms to mean first-order terms from Tm as well as goal terms (in the original definition).

The next subsection is a revised version of Section 2.2.2. I will also introduce some useful theorems about goal terms.

D.2.1 Goal terms

Fix a first-order signature $\Sigma = (\mathbb{B}, \mathbb{S})$ and a constraint language (Tm, Fm, Th) over Σ. In the original paper [Cathcart Burn et al., 2018], the class of well-sorted goal terms $\Delta \vdash G : \rho$, where ρ is a relational sort, is given by these sorting rules:

$$(\textsc{GCst})\ \frac{}{\Delta \vdash c : \rho_c}\ c \in \{\wedge, \vee, \exists_b\} \cup \{\exists_\rho \mid \rho\} \qquad (\textsc{GVar})\ \frac{}{\Delta_1, x : \rho, \Delta_2 \vdash x : \rho}$$

$$(\textsc{GConstr})\ \frac{}{\Delta \vdash \varphi : o}\ \Delta \vdash \varphi : o \in Fm$$

$$(\textsc{GAbs})\ \frac{\Delta, x : \sigma \vdash G : \rho}{\Delta \vdash \lambda x{:}\sigma.G : \sigma \to \rho}\ x \notin \mathtt{dom}(\Delta)$$

$$(\textsc{GAppl})\ \frac{\Delta \vdash G : b \to \rho}{\Delta \vdash G\ N : \rho}\ \Delta \vdash N : b \in Tm$$

$$(\textsc{GAppR})\ \frac{\Delta \vdash G : \rho_1 \to \rho_2 \qquad \Delta \vdash H : \rho_1}{\Delta \vdash G\ H : \rho_2}$$

Throughout the above six rules, b denotes a base sort from $\mathbb{B}$, ρ (with or without subscripts) denotes a relational sort, and σ is either a base sort or a relational sort.

Despite being a subexpression of a goal term, a first-order term $t \in Tm$ is not a goal term according to the definition above. As the defunctionalization algorithm I developed works compositionally, I need a word to refer to not only goal terms but also their subexpressions (excluding subexpressions of elements from $Tm \cup Fm$). Therefore, I will redefine goal terms to encompass first-order terms from Tm:

$$(\textsc{GCst})\ \frac{}{\Delta \vdash c : \rho_c}\ c \in \{\wedge, \vee, \exists_b\} \cup \{\exists_\rho \mid \rho\} \qquad (\textsc{GVar})\ \frac{}{\Delta_1, x : \rho, \Delta_2 \vdash x : \rho}$$

$$(\textsc{GFml})\ \frac{}{\Delta \vdash \varphi : o}\ \Delta \vdash \varphi : o \in Fm \qquad (\textsc{GTerm})\ \frac{}{\Delta \vdash t : b}\ \Delta \vdash t : b \in Tm$$

$$(\textsc{GAbs})\ \frac{\Delta, x : \sigma \vdash G : \rho}{\Delta \vdash \lambda x{:}\sigma.G : \sigma \to \rho}\ x \notin \mathtt{dom}(\Delta)$$

$$(\textsc{GApp})\ \frac{\Delta \vdash G : \sigma \to \rho \qquad \Delta \vdash H : \sigma}{\Delta \vdash G\ H : \rho}$$

As before, throughout the new six rules, b denotes a base sort from $\mathbb{B}$, ρ denotes a relational sort, and σ is either a base sort or a relational sort.

As is true of first-order terms, it is important that Δ contains no conflicts; i.e. no variable is associated with multiple types. Henceforth, it is implicitly assumed that sort environments for goal terms are free of conflicts.

When a goal term t is well-sorted under the sort environment Δ and has sort σ, I write $\Delta \vdash t : \sigma$.

The next three propositions establish the relationship between the original and modified definitions of goal terms.

Proposition D.2.1. *If s is a goal term in the original definition, s can be generated by the new definition. Further, if $\Delta \vdash s : \rho$ in the original definition, then $\Delta \vdash s : \rho$ holds in the new definition as well.*

Proof. The claim is proved by structural induction on s.

For the base case, if s is generated by (GCst) or (GVar), s can be generated by the same rules in the new definition. If s is generated by (GConstr) in the original definition, it can be generated by (GFml) in the new definition. In both cases, the sort is preserved.

For the inductive case, suppose that s is generated by (GAbs) in the original definition. Then it follows from (GAbs) that s is in the form

$$s = \lambda x.G,$$

where G is a goal term in the original definition. Also, if $\Delta, x : \sigma \vdash G : \rho$, then we have $\Delta \vdash \lambda x.G : \sigma \to \rho$. By the inductive hypothesis, G can be generated by the new definition, and $\Delta, x : \sigma \vdash G : \rho$ holds. Hence, by (GAbs) in the new definition, $\Delta \vdash \lambda x.G : \sigma \to \rho$ can be established. Thus, the claim is true in this case.

Consider the case when s is generated by (GAppl) in the original definition. From (GAppl), we know that $s = G\ N$, where G is a goal term and $N \in Tm$. Furthermore, if $\Delta \vdash G : b \to \rho$, then $\Delta \vdash G\ N : \rho$ holds. By the inductive hypothesis, $\Delta \vdash G : b \to \rho$ can be established by the new definition. Also, $\Delta \vdash N : b$ holds in the new definition. Therefore, by (GApp) in the new definition, we obtain

$$(\text{GApp})\ \frac{\Delta \vdash G : b \to \rho \qquad \Delta \vdash N : b}{\Delta \vdash G\ N : \rho}$$

Thus, the claim is true when s is generated by (GAppl).

The case when s is generated by (GAppR) in the original definition can be proved in the same manner as the case when s is created by (GAbs). □

Proposition D.2.2. *If $\Delta \vdash s : \rho$ in the new definition, where ρ is a relational sort, the typing judgement holds in the old definition as well.*

Proof. By structural induction on s. □

Proposition D.2.3. *If A is the set of goal terms in the original definition and B is the set of goal terms in the new definition with relational sorts, then $A = B$ holds.*

Proof. By Proposition D.2.1 and the fact that goal terms in the original definition have relational sorts, we have $A \subseteq B$. Additionally, from Proposition D.2.2, we know $B \subseteq A$. Therefore, by double inclusion, $A = B$. □

Due to Proposition D.2.3, I use the word 'relational goal terms' to mean goal terms in the original definition. Henceforth, I will use the new definition of goal terms.

D.2.2 Properties of goal terms

The first proposition is the goal terms' counterpart of Proposition D.1.2.

Proposition D.2.4. *Given $\Delta \vdash s : \sigma$, the sort of s is unique; that is, we cannot have $\Delta \vdash s : \sigma'$, where $\sigma \neq \sigma'$.*

Proof. The proof proceeds by structural induction on s.

For the base case, if (GCst) or (GVar) is used, the sort of s is uniquely determined by `LSym` or $\mathbb{S}$. If $\Delta \vdash s : \sigma$ is created by (GFml) or (GTerm), the sort of s is uniquely determined due to Proposition D.1.2.

For the inductive case, suppose (GAbs) is used. Hence, we have $s = \lambda x{:}\sigma.G$. Regardless of the sort of s, we can always uniquely determine the sort of x because it is recorded in the lambda abstraction $\lambda x{:}\sigma.G$. Therefore, the left hand side of $\Delta, x : \sigma \vdash G : \rho$ is fixed. It follows from the inductive hypothesis that the sort of G is uniquely determined. Hence, the sort of $\lambda x{:}\sigma.G$ is unique as well.

Finally, if (GApp) is used, we have $s = G\ H$. Since the sorts of G and H under Δ are uniquely determined by the inductive hypothesis, the claim holds for $G\ H$. □

Similarly, the next theorem is the goal terms' counterpart of Theorem D.1.1.

Theorem D.2.1. *If $\Delta \vdash s : \sigma$ holds, where s is a goal term, each symbol in s can be annotated with a unique sort.*

Proof. The proof goes by by structural induction on goal terms.

For the base case, when s is created by (GCst) or (GVar), we can simply apply Proposition D.2.4 since s only contains one symbol. If (GFml) or (GTerm) is used, the claim follows from Theorem D.1.1.

For the inductive case, if s is created by (GAbs), we know $s = \lambda x{:}\sigma.G$. From $\Delta \vdash \lambda x{:}\sigma.G : \sigma \to \rho$, we can uniquely deduce $\Delta, x : \sigma \vdash G : \rho$. By the inductive hypothesis, every symbol in G can be annotated with a unique symbol. Thus, the claim holds in this case.

Finally, if (GApp) is used, we know $s = G\ H$. By Proposition D.2.4, we can uniquely determine the sorts of G and H under Δ; that is, we can uniquely deduce typing judgements $\Delta \vdash G : \rho_1$ and $\Delta \vdash H : \rho_2$. It follows from the inductive hypothesis that every symbol in G and H can be annotated with a unique sort. Therefore, the claim holds for every symbol in $G\ H$. □

The following proposition saves us the need to be concerned about defunctionalizing partially applied instances of functions from $\mathbb{S}$ because they are never strictly partially applied in goal terms.

Proposition D.2.5. *Functions (i.e. constants of arrow sort) from $\mathbb{S}$ cannot be strictly partially applied inside goal terms.*

Proof. Functions from $\mathbb{S}$ are introduced into goal terms by (GFML) and (GTERM). Let s be a first-order term (or formula) introduced by these two rules. Also, let $f \in \mathbb{S}$ be a function and t be a first-order term $f\ t_1\ \cdots t_k$, where $k < \mathbf{ar}(f)$. Hence, t is strictly partially applied. In addition, assume that t cannot be applied to another first-order term. This means that t is maximal with respect to function application. Since (GFML) and (GTERM) require s to be of base sort, s itself cannot be strictly partially applied. Thus, t could only possibly appear (strictly) inside s.

Furthermore, because t is assumed to be maximal with respect to function application, inside s, we cannot have $t\ u$ for some first-order term u. Thus, the only possibility for u being located inside s is that s contains $u\ t$ for some u. However, as indicated by the conclusions in the six typing rules, first-order terms have sorts of order at most 2. Every subexpression of a first-order term is also a first-order term and hence has a sort of order at most 2. Thus, the sort of u has order at most 2; that is, the sort of u looks like $b_1 \to \cdots \to b_n$, where $n > 1$ and $b_i \in \mathbb{B}$ for each $1 \leq i \leq n$. Since u is applied to t, the sort of t must be b_1; that is, t cannot have an arrow sort. Therefore, t cannot appear inside s. This concludes the proof. □

D.3 Type preservation proof

In this section, I prove that in an output of the defunctionalization algorithm, the logic program and the goal formula are well-sorted. Let $\mathcal{P} = (\Delta, P, G)$ be a source monotone problem and $\Sigma = (\mathbb{B}, \mathbb{S})$ be a first-order signature for $\mathcal{P}$. P and G are assumed to be well-sorted. Further, let $\mathcal{P}' = (\Delta', P', G')$ be the result of defunctionalizing $\mathcal{P}$ and $\Sigma' = (\mathbb{B}', \mathbb{S}')$ be a signature for $\mathcal{P}'$.

The first theorem establishes well-sortedness of equations defining Apply_A.

Theorem D.3.1. *Every equation in P'_{Apply} is well-sorted.*

Proof. By (4.8), every equation in P'_{Apply} takes the form

$$\mathit{Apply}_{\sigma'_{n+1}} = \lambda x, y, z.(\exists a_1, \ldots, a_n.x = C^n_X\ a_1\ \cdots\ a_n \wedge z = C^{n+1}_X\ a_1\ \cdots\ a_n\ y), \quad \text{(D.1)}$$

where $X : \sigma_1 \to \cdots \to \sigma_m \to o \in \Delta$, $0 \leq n \leq m - 2$, and $\sigma_i \rightsquigarrow_T \sigma'_i$ for all $1 \leq i \leq n + 1$. In (D.1), the equality between objects of sort **closr** refers to $(=_{\mathbf{closr}})$ declared in $\mathbb{S}'$. The sort of $(=_{\mathbf{closr}})$ is

$$\vdash (=_{\mathbf{closr}}) : \mathbf{closr} \to \mathbf{closr} \to o.$$

Note that I omit $\mathbb{S}'$ from typing judgements whenever its omission does not cause confusion.

From (4.6), we know

$$\Delta' \vdash C^n_X : \sigma'_1 \to \cdots \to \sigma'_n \to \mathbf{closr}$$
$$\Delta' \vdash C^{n+1}_X : \sigma'_1 \to \cdots \to \sigma'_{n+1} \to \mathbf{closr}.$$

Let us denote $\{a_i : \sigma'_i \mid 1 \leq i \leq n\}$ by $\{a_i : \sigma'_i\}$ for brevity. Applying (GApp) repeatedly, we can build the following typing derivations:

$$\frac{\Delta', \{a_i : \sigma'_i\} \vdash \{a_i : \sigma'_i\} \qquad \Delta', \{a_i : \sigma'_i\} \vdash C^n_X : \sigma'_1 \to \cdots \to \sigma'_n \to \mathbf{closr}}{\Delta', \{a_i : \sigma'_i\} \vdash C^n_X\ a_1\ \cdots\ a_n : \mathbf{closr}}$$

$$\frac{\Delta', \{a_i : \sigma'_i\}, y : \sigma'_{n+1} \vdash \{a_i : \sigma'_i\}, y : \sigma'_{n+1} \qquad \Delta', \{a_i : \sigma'_i\}, y : \sigma'_{n+1} \vdash C^{n+1}_X : \sigma'_1 \to \cdots \to \sigma'_{n+1} \to \mathbf{closr}}{\Delta', \{a_i : \sigma'_i\}, y : \sigma'_{n+1} \vdash C^{n+1}_X\ a_1\ \cdots\ a_n\ y : \mathbf{closr}}$$

Hence, for the two disjuncts in (D.1), we have

$$\Delta', \{a_i : \sigma'_i\}, x : \mathbf{closr} : \sigma'_n \vdash (x = C^n_X\ a_1\ \cdots\ a_n) : o$$
$$\Delta', \{a_i : \sigma'_i\}, y : \sigma'_{n+1}, z : \mathbf{closr} : \sigma'_n \vdash (z = C^{n+1}_X\ a_1\ \cdots\ a_n\ y) : o.$$

These two typing judgements yield

$$\Delta', x : \mathbf{closr}, y : \sigma'_{n+1}, z : \mathbf{closr} \vdash (\exists a_1, \ldots, a_n . x = C^n_X\ a_1\ \cdots\ a_n \wedge z = C^{n+1}_X\ a_1\ \cdots\ a_n\ y) : o.$$

Finally, by (GAbs), we obtain

$$\Delta' \vdash \lambda x, y, z . (\exists a_1, \ldots, a_n . x = C^n_X\ a_1\ \cdots\ a_n \wedge z = C^{n+1}_X\ a_1\ \cdots\ a_n\ y) : \mathbf{closr} \to \sigma'_{n+1} \to \mathbf{closr} \to o.$$

Whether $\sigma_{n+1} \in \mathbb{B}$ or $\sigma_{n+1} = \mathbf{closr}$, $\sigma'_{n+1} \in \mathbb{B}'$ holds by the definition of $\mathbb{B}'$. Thus, it is given by (4.6) that

$$\Delta' \vdash \mathit{Apply}_{\sigma'_{n+1}} : \mathbf{closr} \to \sigma'_{n+1} \to \mathbf{closr} \to o.$$

Therefore, the left and right hand sides of (D.1) have the same sort as required. □

The next lemma plays a pivotal role in proving that all equations in P'_{IOMatch} are well-sorted.

Lemma D.3.1. *Let s be a well-sorted source goal term over $\Sigma = (\mathbb{B}, \mathbb{S})$ that contains no lambda abstraction. Also, suppose $App = \{\mathit{Apply}_A : \mathbf{closr} \to A \to \mathbf{closr} \to o \mid A \in \mathbb{B} \cup \{\mathbf{closr}\}\}$ and $IO = \{\mathit{IOMatch}_A : \mathbf{closr} \to A \to o \mid A \in \mathbb{B} \cup \{\mathbf{closr}\}\}$.*

If $\Gamma \vdash s : b$, where $b \in \mathbb{B}$, then $s \rightsquigarrow t$ holds for some goal term t. Furthermore, we have $\Gamma', App, IO \vdash t : b$, where $\Gamma' = \{v : \sigma' \mid v : \sigma \in \Gamma, \sigma \rightsquigarrow_T \sigma'\}$. Here, we use the fact that $\rightsquigarrow_T$ is a function.

Otherwise, if $\Gamma \vdash s : \rho$, where $\rho \notin \mathbb{B}$, then $s \rightsquigarrow^X t$ holds for some goal term t. Furthermore, we have $\Gamma', App, IO, X : \mathbf{closr} \vdash t : o$, where $\Gamma' = \{v : \sigma' \mid v : \sigma \in \Gamma, \sigma \rightsquigarrow_T \sigma'\}$.

Proof. The proof proceeds by structural induction on s.

For the base case, suppose $s \in Fm \cup Tm$. Then the only inference rule that is applicable is (ConstrLan), which gives $s \rightsquigarrow s$. Because s is well-sorted, all free variables in s should be included in Γ. This can be formally proved, but I will not do it here.

Additionally, by Proposition D.1.1, every free variable occurring in first-order terms have base sorts. As $b \rightsquigarrow_T b$ for any $b \in \mathbb{B}$, we have

$$\begin{aligned}\Gamma' &= \{v : \sigma' \mid v : \sigma \in \Gamma, \sigma \rightsquigarrow_T \sigma'\} \\ &= \{u : b \mid u \in \mathtt{FV}(s), u : b \in \Gamma\} \\ &\quad \cup \{v : \sigma' \mid v \notin \mathtt{FV}(s), v : \sigma \in \Gamma, \sigma \rightsquigarrow_T \sigma'\}.\end{aligned}$$

Hence, free variables in s have the same sorts in Γ' as in Γ. As the sort of s depends only on the sorts of free variables in s, we obtain

$$\begin{aligned}&\Gamma' \vdash s : b \\ \therefore\ &\Gamma', \mathit{App}, \mathit{IO} \vdash s : b.\end{aligned}$$

Thus, the claim holds in this case. The case for (Var-Base) can be proved analogously.

Next, consider the case of (Var-Arrow). According to the rule, we have $s = x$, where $\Gamma \vdash x : \rho$ and ρ is a relational arrow sort. (Var-Arrow) yields that $x \rightsquigarrow^X X = x$. As s is well-sorted under Γ, it is given by (GVar) that $x \in \mathtt{dom}(\Gamma)$. Because $\rho \rightsquigarrow_T \mathbf{closr}$ for any relational arrow sort ρ, we have $(x : \mathbf{closr}) \in \Gamma'$. It is straightforward to see that $x : \mathbf{closr}, X : \mathbf{closr} \vdash (X = x) : o$ holds. It thus follows that $\Gamma', \mathit{App}, \mathit{IO}, X : \mathbf{closr} \vdash (X = x) : o$ holds. Therefore, the claim is true in this case. The case for (TopVar) can be proved in the same fashion.

For the inductive case, assume $s = c\ E\ F$, where $c \in \{\wedge, \vee\}$. s is thus defunctionalized by (LogSym). Since s is well-sorted, by (GCst) and (GApp), we have

$$\begin{aligned}&\Gamma \vdash c : o \to o \to o \\ &\Gamma \vdash E : o \\ &\Gamma \vdash F : o.\end{aligned}$$

By the inductive hypothesis, $\Gamma', \mathit{App}, \mathit{IO} \vdash E' : o$ and $\Gamma', \mathit{App}, \mathit{IO} \vdash F' : o$ hold, where $E \rightsquigarrow E'$ and $F \rightsquigarrow F'$. It follows that $\Gamma', \mathit{App}, \mathit{IO} \vdash (c\ E'\ F') : o$.

Next, suppose $s = E\ F$, where $E\ F$ and F have arrow sorts. This case of s is handled by (App) and (App-Arrow). Thus, we have

$$s \rightsquigarrow^X t,$$

where $t = \exists_{\mathbf{closr}} x.(E' \wedge \exists_{\mathbf{closr}} y.(F' \wedge \mathit{Apply}_{\mathbf{closr}}\ x\ y\ X))$ and $E \rightsquigarrow^x E'$ and $F \rightsquigarrow^y F'$. Because both E and F have arrow sorts, the inductive hypothesis gives

$$\begin{aligned}&\Gamma', \mathit{App}, \mathit{IO}, x : \mathbf{closr} \vdash E' : o \\ &\Gamma', \mathit{App}, \mathit{IO}, y : \mathbf{closr} \vdash F' : o.\end{aligned}$$

It is therefore possible to construct a typing derivation tree for $\Gamma', \mathit{App}, \mathit{IO}, X : \mathbf{closr} \vdash t : o$, although I omit it because it takes a lot of space.

The remaining three cases when $s = E\ F$ can be proved analogously. □

Theorem D.3.2. *Every equation from $P'_{IOMatch}$ is well-sorted.*

Proof. By (4.9), each rule in P'_{IOMatch} has the form

$$IOMatch_{\sigma'_m} = \lambda x, x_m.(\exists x_1, \ldots, x_{m-1}.x = C_X^{m-1}\ x_1\ \cdots\ x_{m-1} \wedge F'),$$

where $X = \lambda x_1{:}\sigma_1, \ldots, x_m{:}\sigma_m.F$ is in P. Here, $\mathbf{ar}(X) = m$ and $F \rightsquigarrow F'$. As $\mathbf{ar}(X) = m$, F cannot be a lambda abstraction. Further, equations in P are assumed to be well-sorted. Thus, we obtain

$$\Delta \vdash (\lambda x_1{:}\sigma_1, \ldots, x_m{:}\sigma_m.F) : \sigma_1 \to \cdots \to \sigma_m \to o$$
$$\therefore \Delta, \{x_i : \sigma_i \mid 1 \leq i \leq m\} \vdash F : o.$$

Lemma D.3.1 yields that

$$\Gamma, \{x_i : \sigma'_i \mid 1 \leq i \leq m, \sigma_i \rightsquigarrow_T \sigma'_i\}, \Delta' \vdash F' : o,$$

where $\Gamma = \{X : \sigma' \mid X : \sigma \in \Delta, \sigma \rightsquigarrow_T \sigma'\}$ and Δ' is given by (4.6). It is relatively straightforward to prove that F' does not contain any top-level relational variable symbol from Δ. Therefore, Γ does not affect the sort of F'. Consequently, we obtain

$$\{x_i : \sigma'_i \mid 1 \leq i \leq m, \sigma_i \rightsquigarrow_T \sigma'_i\}, \Delta' \vdash F' : o.$$

It is possible to construct a valid typing derivation tree for

$$\Delta' \vdash \lambda x, x_m.(\exists x_1, \ldots, x_{m-1}.x = C_X^{m-1}\ x_1\ \cdots\ x_{m-1} \wedge F') : \mathbf{closr} \to \sigma'_m \to o.$$

This is consistent with the sort of $IOMatch_{\sigma'_m}$ given by (4.6). Therefore, each equation in P'_{IOMatch} is indeed well-sorted. □

Theorem D.3.3. *Each equation in P' and G' is well-sorted.*

Proof. Well-sortedness of equations in P' follows from Theorem D.3.1 and Theorem D.3.2. As for G', because each $s \in G$ is free of lambda abstractions, by Lemma D.3.1, we have $\Gamma, \Delta' \vdash t : o$, where $s \rightsquigarrow t$ and $\Gamma = \{X : \sigma' \mid X : \sigma \in \Delta\}$. Since G' does not contain any top-level relational variable symbols from Δ, Γ can be removed from the typing judgement. This results in $\Delta' \vdash G' : o$. □

Bibliography

B. Beckert and R. Hähnle. Reasoning and verification: State of the art and current trends. *IEEE Intelligent Systems*, 29(1):20–29, Jan 2014. ISSN 1541-1672. doi: 10.1109/MIS.2014.3. 4

Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven Defunctionalization. In *ICFP*, pages 25–37, 1997. 6

Nikolaj Bjørner, Kenneth L McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. *SMT@ IJCAR*, 20:3–11, 2012. 1, 4, 5

Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. Higher-order program verification as satisfiability modulo theories with algebraic data-types. *CoRR*, abs/1306.5264, 2013. URL http://arxiv.org/abs/1306.5264. 6

Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, pages 24–51. Springer, 2015. 4, 5

Toby Cathcart Burn, C.-H. Luke Ong, and Steven J. Ramsay. Higher-order constrained horn clauses for verification. *PACMPL*, 2(POPL):11:1–11:28, 2018. doi: 10.1145/3158099. URL http://doi.acm.org/10.1145/3158099. 1, 4, 5, 6, 8, 11, 13, 14, 39, 43, 54, 70, 71

Christopher John Hogger. *Essnetials of Logic Programming*. Oxford University Press, 1990. 38

Jerome Jochems. HORS safety verification by reduction to HoCHC. Working paper, 2018. 6, 40

Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and cegar for higher-order model checking. In *ACM SIGPLAN Notices*, volume 46, pages 222–233. ACM, 2011. 43

John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. 38

François Pottier and Nadji Gauthier. Polymorphic Typed Defunctionalization. In *POPL*, pages 89–98, 2004. 6, 25

John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740. ACM, 1972. 6

A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. 38, 39

Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. Mochi: Software model checker for a higher-order functional language. 43

Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. Automating relatively complete verification of higher-order functional programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 75–86, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429081. URL http://doi.acm.org/10.1145/2429069.2429081. 43

Index