AN ABSTRACT OF THE THESIS OF

Spencer Hubbard for the degree of Master of Science in Computer Science presented on June 1, 2016.

Title:

A Formal Foundation for Variational Programming Using the Choice Calculus

Abstract approved: _____

Eric T. Walkingshaw

In this thesis, we present semantic equivalence rules for an extension of the choice calculus and sound operations for an implementation of variational lists. The choice calculus is a calculus for describing variation and the formula choice calculus is an extension with formulas. We prove semantic equivalence rules for the formula choice calculus. Variational lists are functional data structures for representing and computing with variation in lists using the choice calculus. We prove map and bind operations are sound for an implementation of variational lists. These proofs are written and verified in the language of the Coq proof assistant.

A Formal Foundation for Variational Programming Using the Choice Calculus

by

Spencer Hubbard

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 1, 2016
Commencement June 2016

Master of Science thesis of Spencer Hubbard
presented on June 1, 2016.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Spencer Hubbard, Author

ACKNOWLEDGEMENTS

First, I want to thank my parents, Bridget and Roger Hubbard. Thank you for your constant love and support throughout my long but successful academic journey. I love you both.

Next, I want to thank my advisor, Eric Walkingshaw. Thank you for taking me as your student, providing me with funding, and giving me direction with my thesis. I have enjoyed working with you during the short time I have had you as my advisor.

Next, I want to thank my undergraduate mentor, Ron Irving. Thank you for contributing so much to my mathematical maturity and encouraging me to pursue graduate school.

Next, I would like to thank all of the members of my graduate committee, including Martin Erwig, Stephen Redfield, and Maggie Niess. I would also like to thank Martin Erwig and Karl Smeltzer for their feedback on an earlier draft of my thesis.

Finally, I want to thank my roommate Jim Schneidereit and his fianceé Emily Killingbeck for their friendship and sharing in the experience of graduate school with me. I would also like to thank the other friends I have made in Corvallis, including Alex Lepinski and Meara Foster.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (Continued)

Figure Page

Chapter 1 – Introduction

In this thesis, we present semantic equivalence rules for an extension of the choice calculus and sound operations for an implementation of variational lists. The choice calculus is a calculus for describing variation and the formula choice calculus is an extension with formulas. We prove semantic equivalence rules for the formula choice calculus. Variational lists are functional data structures for representing and computing with variation in lists using the choice calculus. We prove map and bind operations are sound for an implementation of variational lists. These proofs are written and verified in the language of the Coq proof assistant.

The choice calculus is a metalanguage for describing variation in an arbitrary object language [Erwig and Walkingshaw, 2011b]. The formula choice calculus is an extension of the choice calculus where choices are labeled with formulas instead of dimensions [Walkingshaw and Ostermann, 2014]. Semantic equivalence rules for the choice calculus have been established by previous work [Walkingshaw, 2013]. In Chapter 2, we establish similar semantic equivalence rules for the formula choice calculus.

Previous work has used the formula choice calculus without proving semantic equivalence rules. For example, a projectional editing model of variational software is based on formula choice calculus and some semantic equivalence rules have been stated—without proof—for this projectional editing model [Walkingshaw and Ostermann, 2014]. As another example, TypeChef is a tool for parsing and type checking C code with preprocessor directives and is based on a model similar to the formula choice calculus [Kenner et al., 2010, Kästner et al., 2011, Walkingshaw et al., 2014]. We provide a formal foundation for previous and future work that use the formula choice calculus.

Variational lists are functional data structures for representing and computing with variation in lists using the choice calculus [Walkingshaw and Erwig, 2012].

Option-lists are an implementation of variational lists as lists of variational optional values [Walkingshaw et al., 2014]. In Chapter 3, we establish sound map[1] and bind[2] operations for option-lists. We also provide a general definition of soundness for operations on variational data structures.

Variational lists are common variational data structures that arise naturally in many variational programming problems. For example, SPLlift is a tool for inter-procedural data-flow analysis on software product lines that uses variational graphs [Bodden et al., 2013]. A variational list could be used for an adjacency list representation of a variational graph [Walkingshaw et al., 2014]. As another example, the CIDE tool [Kästner et al., 2008] and Color Featherweight Java [Kästner et al., 2012] use data structures which are similar to—but less expressive than—option-lists [Walkingshaw et al., 2014]. Previous work has identified a need for foundational research on variational data structures [Walkingshaw et al., 2014]. We provide a formal foundation for option-lists. We also demonstrate a general and principled technique for establishing soundness for operations on variational data structures.

In Appendix A, we provide verified proofs written in the language of the Coq proof assistant [Bertot and Castéran, 2004]. The properties from Chapter 2 are verified in Appendix A.1 and Appendix A.2 and the source code is available online.[3] The properties from Chapter 3 are verified in Appendix A.3 and the source code is also available online.[4]

[1] The map operation is part of the definition of a functor.
[2] The bind operation is part of the definition of a monad.
[3] https://github.com/hubbards/FCC-Coq
[4] https://github.com/hubbards/VP-Coq

Chapter 2 – Formula Choice Calculus

In this chapter, we establish semantic equivalence rules for the formula choice calculus. The choice calculus is a metalanguage for describing variation in an arbitrary object language [Erwig and Walkingshaw, 2011b]. The formula choice calculus is an extension of the choice calculus where choices are labeled with formulas instead of dimensions [Walkingshaw and Ostermann, 2014]. The choice calculus is described in Section 2.1, formulas are described in Section 2.2, the formula choice calculus is described in Section 2.3, and a generalization is discussed in Section 2.4.

2.1 Choice Calculus

Consider an object language X with abstract syntax described by the grammar in Figure 2.1. Atoms represent symbols in the object language and have no internal structure. The tree construct is binary for simplicity. However, the syntax supports encoding of constructs with arbitrary arity, e.g., with right nested trees.

$$
\begin{array}{lcll}
a \in A & & & \textit{Atom} \\
x \in X & ::= & \varepsilon & \textit{Empty} \\
 & | & a\text{-}\prec x, x \succ\text{-} & \textit{Tree}
\end{array}
$$

Figure 2.1: Object language syntax.

The choice calculus is instantiated with the object language X. The abstract syntax of choice calculus expressions is described by the grammar in Figure 2.2. The expressions in the choice construct are called *alternatives* and the dimension is called a *label*.

$$
\begin{array}{rcll}
d \in D & & & \textit{Dimension} \\
e \in E & ::= & \varepsilon & \textit{Empty} \\
& | & a\text{-}\prec e, e \succ\text{-} & \textit{Tree} \\
& | & d\langle e, e \rangle & \textit{Choice}
\end{array}
$$

Figure 2.2: Choice calculus syntax.

A *tag* is either the *left tag* **L** or the *right tag* **R** and a *configuration* is a (total) function from dimensions to tags. Note that a function is total (or everywhere defined) by the mathematically accepted definition of a function and so we omit this hereafter. The set of tags is $T = \{\mathbf{L}, \mathbf{R}\}$ and the set of configurations is $C = D \to T$. The metavariables t and c are used to represent arbitrary tags and configurations, respectively, unless qualified otherwise.

The semantic domain for expressions is the function domain $C \to X$, which is the set of functions from configurations to elements of the object language. The semantic function for expressions is $E[\![\cdot]\!]$, which is defined by the equations in Figure 2.3. Note that we use juxtaposition to indicate function application, as in lambda calculus, e.g., we write $c\ d$ to indicate application of configuration c to dimension d. In the last equation, note that if $c\ d \neq \mathbf{L}$, then $c\ d = \mathbf{R}$. Elements of the object language in the image of $E[\![e]\!]$ are called *variants* of e and application of $E[\![e]\!]$ is called *selection*.

$$
\begin{aligned}
&E[\![\cdot]\!] : E \to C \to X \\
&E[\![\varepsilon]\!]_c = \varepsilon \\
&E[\![a\text{-}\prec e_1, e_2 \succ\text{-}]\!]_c = a\text{-}\prec E[\![e_1]\!]_c, E[\![e_2]\!]_c \succ\text{-} \\
&E[\![d\langle e_1, e_2 \rangle]\!]_c = \begin{cases} E[\![e_1]\!]_c, & \text{if } c\ d = \mathbf{L} \\ E[\![e_2]\!]_c, & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 2.3: Choice calculus semantics.

For example, consider the choice calculus instantiated with the object language of decimal notation. Atoms of this object language are Arabic numerals, e.g., the Arabic numerals 1, 2, and 3 are atoms. Terms of this object language are right nested branches. For convenience, we write the concrete syntax of terms as sequences of Arabic numerals, e.g., 123 and 213 are terms in the concrete syntax which correspond to the terms $1 \prec \varepsilon, 2 \prec \varepsilon, 3 \prec \varepsilon, \varepsilon \succ \succ \succ$ and $2 \prec \varepsilon, 1 \prec \varepsilon, 3 \prec \varepsilon, \varepsilon \succ \succ \succ$, respectively, in the abstract syntax. Suppose d is a dimension. Then $d\langle 12, 21 \rangle 3$ is a choice calculus expression. This expression consists of a sequence with a choice followed by the term 3. The choice is labeled with the dimension d and the alternatives are the terms 12 and 21. Moreover, for each configuration c, the semantics of this expression is the following: (1) if $c\ d = \mathbf{L}$, then the variant 123 is selected, and (2) if $c\ d = \mathbf{R}$, then the variant 213 is selected.

Choices can be extended to support more alternatives. One way to do this is by adding tags to the set of tags and corresponding cases to the semantics of choices. In this way, the number of alternatives is equal to the number of tags. For example, to support three alternatives we add a third tag to the set of tags, extend the syntax of choices with a third alternative, and the semantics of choices with a third case.

2.2 Formulas

In this section we present formulas in tags and dimensions. We describe the syntax and semantics of formulas in Section 2.2.1 and establish rules for deriving formula equivalence in Section 2.2.2.

2.2.1 Denotational Semantics

We begin with a "semantics first" [Erwig and Walkingshaw, 2011a] description of formulas in tags. The set T forms an algebra with prefix unary operator ($\neg$), called *complement*, and infix binary operators ($\vee$) and ($\wedge$), called *join* and *meet*, respectively. These operators are defined by the tables in Figure 2.4. Note that T

is the Boolean algebra with two elements—up to isomorphism—where **L** is "true" and **R** is "false."

t	$\neg\, t$
L	**R**
R	**L**

$\vee$	**L**	**R**
L	**L**	**L**
R	**L**	**R**

$\wedge$	**L**	**R**
L	**L**	**R**
R	**R**	**R**

Figure 2.4: Boolean algebra on tags.

The abstract syntax of formulas is described by the grammar in Figure 2.5. Note that we reuse the symbols for the operators defined in the previous paragraph. The semantic domain for formulas is the function domain $C \to T$, which is the set of functions from configurations to tags. The semantic function for formulas is $F[\![\cdot]\!]$, which is defined by the equations in Figure 2.6.

$$
\begin{array}{lcll}
f \in F & ::= & t & \textit{Tag} \\
& | & d & \textit{Dimension} \\
& | & \neg\, f & \textit{Complement} \\
& | & f \vee f & \textit{Join} \\
& | & f \wedge f & \textit{Meet}
\end{array}
$$

Figure 2.5: Formula syntax.

$$
\begin{aligned}
F[\![\cdot]\!] &: F \to C \to T \\
F[\![t]\!]_c &= t \\
F[\![d]\!]_c &= c\ d \\
F[\![\neg f]\!]_c &= \neg F[\![f]\!]_c \\
F[\![f_1 \vee f_2]\!]_c &= F[\![f_1]\!]_c \vee F[\![f_2]\!]_c \\
F[\![f_1 \wedge f_2]\!]_c &= F[\![f_1]\!]_c \wedge F[\![f_2]\!]_c
\end{aligned}
$$

Figure 2.6: Formula semantics.

2.2.2 Semantic Equivalence

Before we define semantic equivalence, it is useful to recall the following elementary facts: (1) Two functions are equal, by definition, if they have the same domain and codomain and their images agree for all elements in the domain. (2) For any function, the inverse images of elements in the codomain partition the domain. (3) Any partition defines a "canonical" equivalence relation where the parts of the partition correspond to the equivalence classes of the relation.

Let $(\equiv)$ be a binary relation on formulas defined by $f \equiv f'$ if and only if $F[\![f]\!] = F[\![f']\!]$. Note that the relation $(\equiv)$ is defined in terms of function equality. Since $F[\![\cdot]\!]$ is well-defined as a function from formulas to elements of the semantic domain, it follows from earlier remarks that the relation $(\equiv)$ is an equivalence relation. We refer to the relation $(\equiv)$ as *semantic equivalence* and we call formulas f and f' (*semantically*) *equivalent* if $f \equiv f'$.

In the remainder of this section, we establish some syntactic rules for deriving formula equivalence. In practice, formula equivalence can be checked by solving an instance of the satisfiability problem, e.g., using a SAT solver, and this is justified by the syntactic rules. We prove these rules are correct in Appendix A.1.

The formula equivalence rules are stated in Figure 2.7. These rules state that formulas equivalence is reflexive, symmetric, and transitive. These rules follow directly from the definition of an equivalence relation.

$$\textsc{Refl-E}\quad f \equiv f \qquad \textsc{Symm-E}\quad \frac{f \equiv f'}{f' \equiv f} \qquad \textsc{Tran-E}\quad \frac{f_1 \equiv f_2 \qquad f_2 \equiv f_3}{f_1 \equiv f_3}$$

Figure 2.7: Formula equivalence rules.

The formula congruence rules are stated in Figure 2.8. Note that the formula congruence rules are not independent of each other, e.g., Join-Cong can be derived from Join-Cong-L and Join-Cong-R.

$$\textsc{Comp-Cong}\quad \frac{f \equiv f'}{\neg f \equiv \neg f'} \qquad \textsc{Join-Cong}\quad \frac{f_1 \equiv f_1' \qquad f_2 \equiv f_2'}{f_1 \vee f_2 \equiv f_1' \vee f_2'} \qquad \textsc{Join-Cong-L}\quad \frac{f_1 \equiv f_1'}{f_1 \vee f_2 \equiv f_1' \vee f_2} \qquad \textsc{Join-Cong-R}\quad \frac{f_2 \equiv f_2'}{f_1 \vee f_2 \equiv f_1 \vee f_2'}$$

$$\textsc{Meet-Cong}\quad \frac{f_1 \equiv f_1' \qquad f_2 \equiv f_2'}{f_1 \wedge f_2 \equiv f_1' \wedge f_2'} \qquad \textsc{Meet-Cong-L}\quad \frac{f_1 \equiv f_1'}{f_1 \wedge f_2 \equiv f_1' \wedge f_2} \qquad \textsc{Meet-Cong-R}\quad \frac{f_2 \equiv f_2'}{f_1 \wedge f_2 \equiv f_1 \wedge f_2'}$$

Figure 2.8: Formula congruence rules.

The algebraic rules are stated in Figure 2.9. These are the usual rules of Boolean algebra expressed for formula equivalence.

JOIN-COMP $f \vee \neg f \equiv \mathbf{L}$

MEET-COMP $f \wedge \neg f \equiv \mathbf{R}$

JOIN-ID $\mathbf{R} \vee f \equiv f$

MEET-ID $\mathbf{L} \wedge f \equiv f$

JOIN-IDEMP $f \vee f \equiv f$

MEET-IDEMP $f \wedge f \equiv f$

JOIN-COMM $f_1 \vee f_2 \equiv f_2 \vee f_1$

MEET-COMM $f_1 \wedge f_2 \equiv f_2 \wedge f_1$

JOIN-ASSOC $f_1 \vee (f_2 \vee f_3) \equiv (f_1 \vee f_2) \vee f_3$

MEET-ASSOC $f_1 \wedge (f_2 \wedge f_3) \equiv (f_1 \wedge f_2) \wedge f_3$

JOIN-DIST $f_1 \vee (f_2 \wedge f_3) \equiv (f_1 \vee f_2) \wedge (f_1 \vee f_3)$

MEET-DIST $f_1 \wedge (f_2 \vee f_3) \equiv (f_1 \wedge f_2) \vee (f_1 \wedge f_3)$

COMP-JOIN $\neg(f_1 \vee f_2) \equiv \neg f_1 \wedge \neg f_2$

COMP-MEET $\neg(f_1 \wedge f_2) \equiv \neg f_1 \vee \neg f_2$

Figure 2.9: Algebraic rules.

Note that the formula congruence and algebraic rules imply that the set of equivalence classes of formulas is isomorphic to the free Boolean algebra on the set of dimensions D. The operations on equivalence classes are defined in terms of the operations on representative formulas and the formula congruence rules ensure that these operations are well-defined.

As an aside, consider the binary infix operator ($\triangle$) on the set of equivalence classes, called *symmetric difference* (or *exclusive or*). For any representative formulas $f_1, f_2 \in F$, the symmetric difference $f_1 \triangle f_2$ is defined to be the equivalence class with representative formula $(f_1 \wedge \neg f_2) \vee (f_2 \wedge \neg f_1)$. Note that the equivalence classes form a group with law of composition given by symmetric difference. We revisit this idea in Section 2.4.

2.3 Formula Choice Calculus

In this section we present the formula choice calculus. We describe the syntax and semantics of expressions in Section 2.3.1 and establish rules for deriving expression equivalence in Section 2.3.2.

2.3.1 Denotational Semantics

The abstract syntax of expressions is described by the grammar in Figure 2.10. The semantic domain for expressions is the function domain $C \to X$, which is the set of functions from configurations to elements of the object language. The semantic function for expressions is $E[\![\cdot]\!]$, which is defined by the equations in Figure 2.11. In the last equation, note that if $F[\![f]\!]_c \neq \mathbf{L}$, then $F[\![f]\!]_c = \mathbf{R}$.

$$
\begin{array}{rcll}
e \in E & ::= & \varepsilon & \textit{Empty} \\
& | & a\text{-}\!\prec e, e \succ\!\text{-} & \textit{Tree} \\
& | & f\langle e, e \rangle & \textit{Choice}
\end{array}
$$

Figure 2.10: Formula choice calculus syntax.

$$
\begin{aligned}
&E[\![\cdot]\!] : E \to C \to X \\
&E[\![\varepsilon]\!]_c = \varepsilon \\
&E[\![a\text{-}\!\prec e_1, e_2 \succ\!\text{-}]\!]_c = a\text{-}\!\prec E[\![e_1]\!]_c, E[\![e_2]\!]_c \succ\!\text{-} \\
&E[\![f\langle e_1, e_2 \rangle]\!]_c = \begin{cases} E[\![e_1]\!]_c, & \text{if } F[\![f]\!]_c = \mathbf{L} \\ E[\![e_2]\!]_c, & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 2.11: Formula choice calculus semantics.

For example, consider the formula choice calculus instantiated with the same object language as before, i.e., the object language of decimal notation. Suppose

d_1 and d_2 are dimensions. Then $(d_1 \vee d_2)\langle 12, 21\rangle 3$ is a formula choice calculus expression. This expression consists of a similar sequence as before, except the choice is labeled with the formula $d_1 \vee d_2$. Moreover, for each configuration c, the semantics of this expression is the following: (1) if $c\ d_1 = \mathbf{L}$ or $c\ d_2 = \mathbf{L}$, then the variant 123 is selected, and (2) if $c\ d_1 = \mathbf{R}$ and $c\ d_2 = \mathbf{R}$, then the variant 213 is selected.

2.3.2 Semantic Equivalence

Semantic equivalence for expressions is defined in the same way as it is for formulas. Let $(\equiv)$ be a binary relation on expressions defined by $e \equiv e'$ if and only if $E[\![e]\!] = E[\![e']\!]$. By the same reasoning as before, the relation $(\equiv)$ is an equivalence relation. We refer to the relation $(\equiv)$ as *semantic equivalence* and we call expressions e and e' (*semantically*) *equivalent* if $e \equiv e'$.

In the remainder of this section, we establish some syntactic rules for deriving expression equivalence. We prove these rules are correct in Appendix A.2. However, we do not prove these rules are complete, i.e., if $e \equiv e'$, then there is a derivation by these rules. Proving the completeness of a minimal set of rules is left to future work.

All of the rules stated in this section can be proved directly from the definition of expression equivalence. However, this often leads to duplication of logic among the proofs. Moreover, these proofs are not very insightful. To avoid duplication and to describe relationships among rules, we derive some rules from others—rather than directly from definitions.

The expression equivalence rules are stated in Figure 2.12. These rules state that expression equivalence is reflexive, symmetric, and transitive. These rules follow directly from the definition of an equivalence relation.

$$\textsc{Refl-E}\quad e \equiv e \qquad \textsc{Symm-E}\quad \frac{e \equiv e'}{e' \equiv e} \qquad \textsc{Tran-E}\quad \frac{e_1 \equiv e_2 \qquad e_2 \equiv e_3}{e_1 \equiv e_3}$$

Figure 2.12: Expression equivalence rules.

The choice transposition rule is stated in Figure 2.13. This rule states that the semantics of a choice is invariant under transposition of its alternatives and complementation of its label. This rule allows us to derive some rules from others and it is used extensively in these derivations. We often omit details of applying this rule for brevity. We prove this rule directly from the definition of expression equivalence, see Appendix A.2.

$$\textsc{Chc-Tran}\quad f\langle e_1, e_2\rangle \equiv \neg\, f\langle e_2, e_1\rangle$$

Figure 2.13: Choice transposition rule.

The object congruence rules are stated in Figure 2.14. Note that these rules are not independent of each other. For example, Obj-Cong can be derived from the other two rules. These rules do not refer to formula choices directly and so they do not differ from the corresponding rules for dimension choices in a meaningful way. For completeness, we include proofs of these rules in Appendix A.2.

$$\textsc{Obj-Cong}\quad \frac{e_1 \equiv e_1' \qquad e_2 \equiv e_2'}{a\text{-}\prec e_1, e_2 \succ\text{-} \equiv a\text{-}\prec e_1', e_2' \succ\text{-}} \qquad \textsc{Obj-Cong-l}\quad \frac{e_1 \equiv e_1'}{a\text{-}\prec e_1, e_2 \succ\text{-} \equiv a\text{-}\prec e_1', e_2 \succ\text{-}} \qquad \textsc{Obj-Cong-r}\quad \frac{e_2 \equiv e_2'}{a\text{-}\prec e_1, e_2 \succ\text{-} \equiv a\text{-}\prec e_1, e_2' \succ\text{-}}$$

Figure 2.14: Object congruence rules.

The choice congruence rules are stated in Figure 2.15. Note that these rules are not independent of each other. For example, we show Chc-Cong directly from the other three rules in the proof of Theorem 2.1. Of course, we could also derive the other three rules directly from Chc-Cong. The choice congruence rule for labels Chc-Cong-F is used in derivations of some other rules. We often omit details of applying this rule for brevity.

$$\textsc{Chc-Cong}\quad \frac{e_1 \equiv e_1' \qquad e_2 \equiv e_2' \qquad f \equiv f'}{f\langle e_1, e_2\rangle \equiv f'\langle e_1', e_2'\rangle}$$

$$\textsc{Chc-Cong-F}\quad \frac{f \equiv f'}{f\langle e_1, e_2\rangle \equiv f'\langle e_1, e_2\rangle} \qquad \textsc{Chc-Cong-L}\quad \frac{e_1 \equiv e_1'}{f\langle e_1, e_2\rangle \equiv f\langle e_1', e_2\rangle} \qquad \textsc{Chc-Cong-R}\quad \frac{e_2 \equiv e_2'}{f\langle e_1, e_2\rangle \equiv f\langle e_1, e_2'\rangle}$$

Figure 2.15: Choice congruence rules.

Theorem 2.1. The choice congruence rules hold for all formulas $f, f' \in F$ and expressions $e_1, e_1', e_2, e_2' \in E$.

Proof. First, we show Chc-Cong-F and Chc-Cong-L directly from the definition of expression equivalence, see Appendix A.2. Next, we show Chc-Cong-R by deriving the consequent from the antecedent and Chc-Cong-L. The derivation is as follows.

$$\begin{aligned} f\langle e_1, e_2\rangle &\equiv \neg\, f\langle e_2, e_1\rangle \\ &\equiv \neg\, f\langle e_2', e_1\rangle \\ &\equiv f\langle e_1, e_2'\rangle \end{aligned}$$

Finally, Chc-Cong follows directly from the other three rules. ■

Observe that converses of the congruence rules for choices do not hold, e.g., in general $f\langle e_1, e_2\rangle \equiv f'\langle e_1, e_2\rangle$ is not sufficient for $f \equiv f'$ and $f\langle e_1, e_2\rangle \equiv f\langle e_1', e_2\rangle$ is not sufficient for $e_1 \equiv e_1'$. For a counterexample to the converse of Chc-Cong-F, consider $\mathbf{L}\langle a, a\rangle \equiv a \equiv \mathbf{R}\langle a, a\rangle$ but $\mathbf{L} \not\equiv \mathbf{R}$. For a counterexample to the converse of Chc-Cong-L, consider $\mathbf{R}\langle a, a\rangle \equiv a \equiv \mathbf{R}\langle a', a\rangle$ but $a \not\equiv a'$ if $a \neq a'$. There is a similar counterexample to the converse of Chc-Cong-R.

The choice simplification rules are stated in Figure 2.16. These rules describe certain cases where the semantics of a choice is equivalent to one of its alternatives.

Chc-Idemp	Chc-L	Chc-R
$f\langle e, e\rangle \equiv e$	$\mathbf{L}\langle e_1, e_2\rangle \equiv e_1$	$\mathbf{R}\langle e_1, e_2\rangle \equiv e_2$

Figure 2.16: Choice simplification rules.

Theorem 2.2. The choice simplification rules hold for all formulas $f \in F$ and expressions $e, e_1, e_2 \in E$.

Proof. First, we show Chc-Idemp and Chc-L directly from the definition of expression equivalence, see Appendix A.2. Next, we show Chc-R by Chc-L and $\neg\, \mathbf{R} \equiv \mathbf{L}$, which can be shown from the formula congruence and algebraic rules, see Appendix A.1. The derivation of Chc-R is as follows.

$$\begin{aligned} \mathbf{R}\langle e_1, e_2\rangle &\equiv \neg\, \mathbf{R}\langle e_2, e_1\rangle \\ &\equiv \mathbf{L}\langle e_2, e_1\rangle \\ &\equiv e_2 \end{aligned}$$

■

The formula choice rules are stated in Figure 2.17. Redundant alternatives in nested choices can be eliminated while preserving semantics by applying these rules from left-to-right.

Chc-Join	Chc-Meet
$f_1\langle e_1, f_2\langle e_1, e_2\rangle\rangle \equiv (f_1 \vee f_2)\langle e_1, e_2\rangle$	$f_1\langle f_2\langle e_1, e_2\rangle, e_2\rangle \equiv (f_1 \wedge f_2)\langle e_1, e_2\rangle$
Chc-Join-Comp	**Chc-Meet-Comp**
$f_1\langle e_1, f_2\langle e_2, e_1\rangle\rangle \equiv (f_1 \vee \neg f_2)\langle e_1, e_2\rangle$	$f_1\langle f_2\langle e_2, e_1\rangle, e_2\rangle \equiv (f_1 \wedge \neg f_2)\langle e_1, e_2\rangle$

Figure 2.17: Formula choice rules.

Theorem 2.3. The formula choice rules hold for all formulas $f_1, f_2 \in F$ and expressions $e_1, e_2 \in E$.

Proof. First, we show Chc-Join directly from the definition of expression equivalence, see Appendix A.2. Next, we show Chc-Meet by Chc-Join and Comp-Meet. The derivation of Chc-Meet is as follows.

$$\begin{aligned} f_1\langle f_2\langle e_1, e_2\rangle, e_2\rangle &\equiv \neg f_1\langle e_2, \neg f_2\langle e_2, e_1\rangle\rangle \\ &\equiv (\neg f_1 \vee \neg f_2)\langle e_2, e_1\rangle \\ &\equiv \neg(f_1 \wedge f_2)\langle e_2, e_1\rangle \\ &\equiv (f_1 \wedge f_2)\langle e_1, e_2\rangle \end{aligned}$$

Finally, we show Chc-Join-Comp and Chc-Meet-Comp by Chc-Join and Chc-Meet, respectively. The derivation of Chc-Join-Comp is as follows.

$$\begin{aligned} f_1\langle e_1, f_2\langle e_2, e_1\rangle\rangle &\equiv f_1\langle e_1, \neg f_2\langle e_1, e_2\rangle\rangle \\ &\equiv (f_1 \vee \neg f_2)\langle e_1, e_2\rangle \end{aligned}$$

The derivation of Chc-Meet-Comp is as follows.

$$\begin{aligned} f_1\langle f_2\langle e_2, e_1\rangle, e_2\rangle &\equiv f_1\langle \neg f_2\langle e_1, e_2\rangle, e_2\rangle \\ &\equiv (f_1 \wedge \neg f_2)\langle e_1, e_2\rangle \end{aligned}$$

■

The choice merge rules are stated in Figure 2.18. Unselectable alternatives in nested choices can be eliminated while preserving semantics by applying these rules from left-to-right. Note that these rules are not independent of each other. For example, we derive CC-Merge from the other two choice merge rules in Theorem 2.4. Alternatively, we could derive CC-Merge-L and CC-Merge-R from CC-Merge and Chc-Idemp.

CC-Merge

$$f\langle f\langle e_1, e_2\rangle, f\langle e_3, e_4\rangle\rangle \equiv f\langle e_1, e_4\rangle$$

CC-Merge-L

$$f\langle f\langle e_1, e_2\rangle, e_3\rangle \equiv f\langle e_1, e_3\rangle$$

CC-Merge-R

$$f\langle e_1, f\langle e_2, e_3\rangle\rangle \equiv f\langle e_1, e_3\rangle$$

Figure 2.18: Choice merge rules.

Theorem 2.4. The choice merge rules hold for all formulas $f_1, f_2 \in F$, and expressions $e_1, e_2, e_3, e_4 \in E$.

Proof. First, we show CC-Merge-L directly from the definition of expression equivalence, see Appendix A.2. Next, we show CC-Merge-R by CC-Merge-L. The derivation of CC-Merge-R is as follows.

$$\begin{aligned} f\langle e_1, f\langle e_2, e_3\rangle\rangle &\equiv \neg\, f\langle \neg\, f\langle e_3, e_2\rangle, e_1\rangle \\ &\equiv \neg\, f\langle e_3, e_1\rangle \\ &\equiv f\langle e_1, e_3\rangle \end{aligned}$$

Finally, CC-Merge follows directly from the other two rules. ■

The object-choice commutation rule is stated in Figure 2.19. Although this rule refers to formula choices directly, it does not differ from the corresponding rule for dimension choices in a meaningful way. For completeness, we include a proof of this rule in Appendix A.2.

CO-Swap

$$f\langle a{\prec}e_1, e_2{\succ}, a{\prec}e_3, e_4{\succ}\rangle \equiv a{\prec}f\langle e_1, e_3\rangle, f\langle e_2, e_4\rangle{\succ}$$

Figure 2.19: Object-choice commutation rule.

The choice-choice commutation rules are stated in Figure 2.20. Observe that we derived CC-Swap-L from CC-Swap and Chc-Idemp in the proof of Theorem 2.5. Alternatively, we could derive CC-Swap from the other two choice-choice commutation rules, and the choice merge rules.

CC-Swap

$$f_1\langle f_2\langle e_1, e_2\rangle, f_2\langle e_3, e_4\rangle\rangle \equiv f_2\langle f_1\langle e_1, e_3\rangle, f_1\langle e_2, e_4\rangle\rangle$$

CC-Swap-L

$$f_1\langle f_2\langle e_1, e_2\rangle, e_3\rangle \equiv f_2\langle f_1\langle e_1, e_3\rangle, f_1\langle e_2, e_3\rangle\rangle$$

CC-Swap-R

$$f_1\langle e_1, f_2\langle e_2, e_3\rangle\rangle \equiv f_2\langle f_1\langle e_1, e_2\rangle, f_1\langle e_1, e_3\rangle\rangle$$

Figure 2.20: Choice-choice commutation rules.

Theorem 2.5. The choice-choice commutation rules hold for all formulas $f_1, f_2 \in F$, and expressions $e_1, e_2, e_3, e_4 \in E$.

Proof. First, we show CC-Swap directly from the definition of expression equivalence, see Appendix A.2. Next, we show CC-Swap-L by CC-Swap and Chc-Idemp. The derivation of CC-Swap-L is as follows.

$$\begin{aligned} f_1\langle f_2\langle e_1, e_2\rangle, e_3\rangle &\equiv f_1\langle f_2\langle e_1, e_2\rangle, f_2\langle e_3, e_3\rangle\rangle \\ &\equiv f_2\langle f_1\langle e_1, e_3\rangle, f_1\langle e_2, e_3\rangle\rangle \end{aligned}$$

Finally, we show CC-Swap-r by CC-Swap-l. The derivation of CC-Swap-r is as follows.

$$\begin{aligned} f_1\langle e_1, f_2\langle e_2, e_3\rangle\rangle &\equiv \neg\, f_1\langle f_2\langle e_2, e_3\rangle, e_1\rangle \\ &\equiv f_2\langle \neg\, f_1\langle e_2, e_1\rangle, \neg\, f_1\langle e_3, e_1\rangle\rangle \\ &\equiv f_2\langle f_1\langle e_1, e_2\rangle, f_1\langle e_1, e_3\rangle\rangle \end{aligned}$$ ■

2.4 Generalizations

Similar to dimension choices, formula choices can be extended to support more alternatives by adding tags to the set of tags and corresponding cases to the semantics of choices. However, the set of tags must also form a Boolean algebra—unlike the corresponding extension for dimension choices. Since any non-trivial Boolean algebra has an even number of elements, it follows that formula choices cannot be extended with an odd number of alternatives in this way.

To see why any non-trivial Boolean algebra has an even number of elements, note that any Boolean algebra is a group with law of composition given by symmetric difference. In such a group, every element is its own inverse. This means the order of a non-identity element is two, which means the order of the group is a multiple of two by Lagrange's theorem [Dummit and Foote, 2004, p. 89].

Notwithstanding this limitation, formula choices can be extended with an even number of alternatives. Consider the following example of extending formula choices with four alternatives. The set $T = \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}\}$ together with the operators defined by the tables in Figure 2.21 is the Boolean algebra with four elements—up to isomorphism—where **1** is "true" and **4** is "false." The syntax and semantics of formulas is defined in the same way as before. For expressions, we extend the syntax and semantics of choices in a similar way to the example extension from Section 2.1, albeit with four alternatives and cases instead of three.

t	$\neg\, t$
1	**4**
2	**3**
3	**2**
4	**1**

$\vee$	**1**	**2**	**3**	**4**
1	**1**	**1**	**1**	**1**
2	**1**	**2**	**1**	**2**
3	**1**	**1**	**3**	**3**
4	**1**	**2**	**3**	**4**

$\wedge$	**1**	**2**	**3**	**4**
1	**1**	**2**	**3**	**4**
2	**2**	**2**	**4**	**4**
3	**3**	**4**	**3**	**4**
4	**4**	**4**	**4**	**4**

Figure 2.21: Boolean algebra on four tags.

The equivalence rules for choices must be modified as well. The choice merge rule CC-Merge is replaced with the following rule:

$$\frac{e_i = f\langle e_{i1}, e_{i2}, e_{i3}, e_{i4}\rangle \qquad \forall\, i \in \{1,2,3,4\}}{f\langle e_1, e_2, e_3, e_4\rangle \equiv f\langle e_{11}, e_{22}, e_{33}, e_{44}\rangle}$$

Note that there are four premises of this rule. The choice-choice commutation rule CC-Swap is replaced with the following rule:

$$\frac{e_i = f'\langle e_{i1}, e_{i2}, e_{i3}, e_{i4}\rangle \qquad e'_i = f\langle e_{1i}, e_{2i}, e_{3i}, e_{4i}\rangle \qquad \forall\, i \in \{1,2,3,4\}}{f\langle e_1, e_2, e_3, e_4\rangle \equiv f'\langle e'_1, e'_2, e'_3, e'_4\rangle}$$

Note that there are eight premises of this rule. Similar modifications must be made to other rules.

Chapter 3 – Variational Programming

In this chapter, we establish sound map[1] and bind[2] operations for option-lists. We also provide a general definition of soundness for operations on variational data structures. Variational lists are functional data structures for representation and computation with variation in lists using the choice calculus [Walkingshaw and Erwig, 2012]. Option-lists are an efficient implementation of variational lists which support sharing of list elements among variant lists [Walkingshaw et al., 2014]. Variational data structures are described in Section 3.1 and variational lists are described in Section 3.2. A sound map operation is established in Section 3.2.1 and a sound bind operation is established in Section 3.2.2.

3.1 Variational Data Structures

A *variational value* of type X is a value of type $V\,X$, which is the choice calculus instantiated with type X. The abstract syntax of variational values is described for an arbitrary type X by the grammar in Figure 3.1. Note that formula choices are used. However, formula choices could be replaced by dimension choices—along with appropriate changes to selection—and the results of this chapter would still hold.

$$\begin{array}{rcll} x \in X & & & \textit{Value} \\ v \in V\,X & ::= & x & \\ & \mid & f\langle v, v\rangle & \textit{Choice} \end{array}$$

Figure 3.1: Variational value syntax.

[1] The map operation is part of the definition of a functor.
[2] The bind operation is part of the definition of a monad.

The selection operation for variational values is given by the function *sel*, which is defined for an arbitrary type X by the equations in Figure 3.2. In the last equation, note that if $F[\![f]\!]_c \neq \mathbf{L}$, then $F[\![f]\!]_c = \mathbf{R}$. This function selects a variant from a variational value.

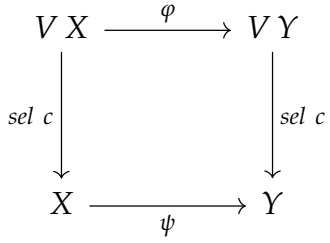
$$
\begin{aligned}
&\mathit{sel} : C \to V\,X \to X \\
&\mathit{sel}\ c\ x \qquad\quad = x \\
&\mathit{sel}\ c\ f\langle v_1, v_2\rangle = \begin{cases} \mathit{sel}\ c\ v_1, & \text{if } F[\![f]\!]_c = \mathbf{L} \\ \mathit{sel}\ c\ v_2, & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.2: Selection operation for variational values.

For example, consider variational integers. A variational integer is a variational value with integer variants. Suppose d_1 and d_2 are dimensions. Then $d_1\langle 1, d_2\langle 2, 3\rangle\rangle$ is a variational integer. This variational integer consists of a choice. The label of the choice is the dimension d_1 and the alternatives are the integer 1 and the nested choice $d_2\langle 2, 3\rangle$. The label of the nested choice is the dimension d_2 and the alternatives are the integers 2 and 3. Moreover, for each configuration c, the image of this variational integer under the partially applied function *sel* c is the following: (1) if $c\ d_1 = \mathbf{L}$, then the image is the integer 1, (2) if $c\ d_1 = \mathbf{R}$ and $c\ d_2 = \mathbf{L}$, then the image is the integer 2, and (3) if $c\ d_1 = \mathbf{R}$ and $c\ d_2 = \mathbf{R}$, then the image is the integer 3.

A function $\varphi : V\,X \to V\,Y$ is *sound* at a configuration $c \in C$ and with respect to a function $\psi : X \to Y$ if the equality $\mathit{sel}\ c \circ \varphi = \psi \circ \mathit{sel}\ c$ holds. Equivalently, the

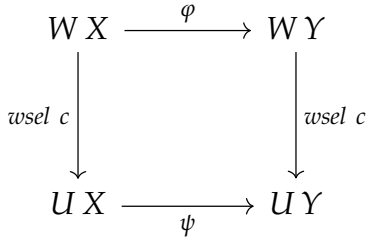
following diagram commutes:



Moreover, a function φ is *sound* with respect to a function ψ if φ is sound at every configuration with respect to ψ. Informally, a (variational) function is sound at a configuration and with respect to a (plain) function if for each element in the domain of the variational function, (1) the image under the plain function of the variant selected from the element is equivalent to (2) the variant selected from the image of the element under the variational function.

In general, a data structure is given by a type constructor $U\,(\cdot)$ and the corresponding *variational data structure* is $V \circ U$. A data structure $W\,(\cdot)$ *implements* a variational data structure $V \circ U$ with a selection operation given by a polymorphic function $wsel : C \to W\,X \to U\,X$ with type variable X.

Suppose $W\,(\cdot)$ implements $V \circ U$ with selection given by $wsel$. Then a function $\varphi : W\,X \to W\,Y$ is *sound* at a configuration $c \in C$ and with respect to a function $\psi : U\,X \to U\,Y$ if the equality $wsel\ c \circ \varphi = \psi \circ wsel\ c$ holds. Equivalently, the following diagram commutes:



Moreover, a function φ is *sound* with respect to a function ψ if φ is sound at every configuration with respect to ψ.

3.2 Variational Lists

An *option* (or *optional value*) of type X is a value of the lifted type $X_\perp = X \cup \{\perp\}$ where $\perp \notin X$. The symbol $\perp$ is called the *bottom value*. Note that we use the symbol $\perp$ to denote the bottom values of both $X_\perp$ and $Y_\perp$. A *variational option* of type X is a value of type $V\,X_\perp$.

A *list* of elements of type X is a list of type $L\,X$. The standard abstract syntax for lists is described for an arbitrary type X by the grammar in Figure 3.3. A *variational list* of elements of type X is a value of type $V\,(L\,X)$. An *option-list* of elements of type X is a list of type $L\,(V\,X_\perp)$. For convenience, we write this as $O\,X$, i.e., $O\,X = L\,(V\,X_\perp)$.

$$\begin{array}{rcll} x \in X & & & \textit{Element} \\ l \in L\,X & ::= & \varepsilon & \textit{Nil} \\ & | & x :: l & \textit{Cons} \end{array}$$

Figure 3.3: List syntax.

Option-lists are an implementation of variational lists with selection given by the function *osel*, which is defined for an arbitrary type X by the equations in Figure 3.4. In the last equation, note that if $\mathit{sel}\ c\ v \neq x$, then $\mathit{sel}\ c\ v = \perp$ since $v \in V\,X_\perp$ and so $\mathit{sel}\ c\ v \in X_\perp$. This function selects a variant list from an option-list by selecting variant elements from the option-list elements and discarding those which are the bottom value.

$$
\begin{aligned}
&\mathit{osel} : C \to O\,X \to L\,X \\
&\mathit{osel}\ c\ \varepsilon \qquad\quad = \varepsilon \\
&\mathit{osel}\ c\ (v :: l) = \begin{cases} x :: \mathit{osel}\ c\ l, & \text{if } \mathit{sel}\ c\ v = x \\ \mathit{osel}\ c\ l, & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.4: Selection operation for option-list.

For example, consider lists and option-lists of integers. For convenience, we write the concrete syntax of lists as comma separated sequences of elements delimited by square brackets, e.g., $[1,2,3]$ and $[2,4]$ are lists of integers in the concrete syntax which correspond to the lists of integers $1 :: 2 :: 3 :: \varepsilon$ and $2 :: 4 :: \varepsilon$, respectively, in the abstract syntax. An option list of integers is a list where the elements are variational options and the non-bottom variants are integers. Suppose d is a dimension. Then $[d\langle 1, \bot\rangle, 2, d\langle 3,4\rangle]$ is an option-list of integers. Moreover, for each configuration c, the image of this option-list under the partially applied function *osel c* is the following: (1) if $c\ d = \mathbf{L}$, then the image is the variant list $[1,2,3]$ and (2) if $c\ d = \mathbf{R}$, then the image is the variant list $[2,4]$.

3.2.1 Map Operation

The list type constructor $L\,(\cdot)$ together with some map operation form a functor [Wadler, 1992, Wadler, Philip, 1992, Wadler, 1995]. The standard map operation for lists is given by the function *map*, which is defined for arbitrary types X and Y by the equations in Figure 3.5. This function maps a function over the elements of a list.

$$
\begin{aligned}
&\mathit{map} : (X \to Y) \to L\,X \to L\,Y \\
&\mathit{map}\ \varphi\ \varepsilon \qquad\ = \varepsilon \\
&\mathit{map}\ \varphi\ (x :: l) = \varphi\ x :: \mathit{map}\ \varphi\ l
\end{aligned}
$$

Figure 3.5: Map operation for list.

It can be shown that the option-list type constructor $O\,(\cdot)$ together with some map operation form a functor. The map operation for option-lists is given by the function *omap*, which is defined for arbitrary types X and Y by the equations in Figure 3.6. This function maps a function over the elements of an option-list.

$$
\begin{aligned}
&\mathit{omap} : (X \to Y) \to O\,X \to O\,Y \\
&\mathit{omap}\ \varphi\ \varepsilon \qquad\ = \varepsilon \\
&\mathit{omap}\ \varphi\ (v :: l) = \mathit{hmap}\ \varphi\ v :: \mathit{omap}\ \varphi\ l
\end{aligned}
$$

Figure 3.6: Map operation for option-lists.

The function *hmap* is defined for arbitrary types X and Y by the equations in Figure 3.7. This function maps a function over the variants of a variational option.

$$
\begin{aligned}
&\mathit{hmap} : (X \to Y) \to V\,X_\perp \to V\,Y_\perp \\
&\mathit{hmap}\ \varphi\ \perp \qquad\ = \perp \\
&\mathit{hmap}\ \varphi\ x \qquad\ = \varphi\ x \\
&\mathit{hmap}\ \varphi\ f\langle v_1, v_2\rangle = f\langle \mathit{hmap}\ \varphi\ v_1, \mathit{hmap}\ \varphi\ v_2\rangle
\end{aligned}
$$

Figure 3.7: *hmap* function definition.

We use Lemma 3.1 and Lemma 3.2 to prove soundness of the map operation for option-lists. Informally, the first lemma says that mapping a function over a variant which is the bottom value does not change the value of the variant and the second lemma says that mapping a function over a variant which is a (non-bottom) value results in application of the function to that value.

Lemma 3.1. For all configurations $c \in C$, functions $\varphi : X \to Y$, and variational options $v \in V\ X_\perp$, if $sel\ c\ v = \perp$, then $sel\ c\ (hmap\ \varphi\ v) = \perp$.

Proof. We will use structural induction on v.

For the first base case, suppose $v = \perp$. Then by the definitions of *hmap* and *sel* we have $sel\ c\ (hmap\ \varphi\ \perp) = \perp$ and we are done. For the second base case, suppose $v = x$ for some value $x \in X$. Then by the definition of *sel* and the hypothesis of the lemma we have $x = sel\ c\ x = \perp$, which is a contradiction and so this case is impossible.

For the induction step, suppose $v = f\langle v_1, v_2\rangle$ for some formula $f \in F$ and variational options $v_1, v_2 \in V\ X_\perp$ with the following inductive hypotheses:

$$sel\ c\ v_1 = \perp \implies sel\ c\ (hmap\ \varphi\ v_1) = \perp$$
$$sel\ c\ v_2 = \perp \implies sel\ c\ (hmap\ \varphi\ v_2) = \perp$$

Notice that $F[\![f]\!]_c$ is either **L** or **R**.

If $F[\![f]\!]_c = \mathbf{L}$, then by the definition of *sel* and the hypothesis of the lemma we have $sel\ c\ v_1 = sel\ c\ f\langle v_1, v_2\rangle = \perp$. By applying this result to our first inductive hypothesis, we have the following derivation by the definitions of *hmap* and *sel*.

$$\begin{aligned} sel\ c\ (hmap\ \varphi\ f\langle v_1, v_2\rangle) &= sel\ c\ f\langle hmap\ \varphi\ v_1, hmap\ \varphi\ v_2\rangle \\ &= sel\ c\ (hmap\ \varphi\ v_1) \\ &= \perp \end{aligned}$$

The case where $F[\![f]\!]_c = \mathbf{R}$ follows by similar reasoning, see Appendix A.3. ■

Lemma 3.2. For all configurations $c \in C$, functions $\varphi : X \to Y$, and variational options $v \in V\ X_\perp$, if $sel\ c\ v = x$ for some value $x \in X$, then $sel\ c\ (hmap\ \varphi\ v) = \varphi\ x$.

Proof. We will use structural induction on v.

For the first base case, suppose $v = \perp$. Then by the definition of *sel* and the hypothesis of the lemma we have $\perp = sel\ c\ \perp = x$, which is a contradiction and so this case is impossible. For the second base case, suppose $v = x'$ for some value $x' \in X$. Then by the definition of *sel* and the hypothesis of the lemma we have $x' = sel\ c\ x' = x$. By the definitions of *hmap* and *sel*, this means $sel\ c\ (hmap\ \varphi\ x') = \varphi\ x' = \varphi\ x$.

For the induction step, suppose $v = f\langle v_1, v_2\rangle$ for some formula $f \in F$ and variational options $v_1, v_2 \in V\ X_\perp$ with the following inductive hypotheses:

$$sel\ c\ v_1 = x \implies sel\ c\ (hmap\ \varphi\ v_1) = x$$
$$sel\ c\ v_2 = x \implies sel\ c\ (hmap\ \varphi\ v_2) = x$$

Notice that $F[\![f]\!]_c$ is either **L** or **R**.

If $F[\![f]\!]_c = \mathbf{L}$, then by the definition of *sel* and the hypothesis of the lemma we have $sel\ c\ v_1 = sel\ c\ f\langle v_1, v_2\rangle = x$. By applying this result to our first inductive hypothesis, we have the following derivation by the definitions of *hmap* and *sel*.

$$\begin{aligned} sel\ c\ (hmap\ \varphi\ f\langle v_1, v_2\rangle) &= sel\ c\ f\langle hmap\ \varphi\ v_1, hmap\ \varphi\ v_2\rangle \\ &= sel\ c\ (hmap\ \varphi\ v_1) \\ &= x \end{aligned}$$

The case where $F[\![f]\!]_c = \mathbf{R}$ follows by similar reasoning, see Appendix A.3. ■

We use the definition of soundness for option-lists to define soundness of the map operation for option-lists. The map operation for option-lists is *sound* if for each function $\varphi : X \to Y$, the function $omap\ \varphi$ is sound with respect to the function $map\ \varphi$. This is stated formally by Theorem 3.3. A machine verified version of the proof of this theorem is presented in Appendix A.3.

Theorem 3.3. For all configurations $c \in C$ and functions $\varphi : X \to Y$, the equality $\mathit{osel}\ c \circ \mathit{omap}\ \varphi = \mathit{map}\ \varphi \circ \mathit{osel}\ c$ holds. Equivalently, the following diagram commutes:

$$\begin{array}{ccc} O\,X & \xrightarrow{\mathit{omap}\ \varphi} & O\,Y \\ {\scriptstyle \mathit{osel}\ c}\downarrow & & \downarrow{\scriptstyle \mathit{osel}\ c} \\ L\,X & \xrightarrow[\mathit{map}\ \varphi]{} & L\,Y \end{array}$$

Proof. We will show that the equality $(\mathit{osel}\ c \circ \mathit{omap}\ \varphi)\ l = (\mathit{map}\ \varphi \circ \mathit{osel}\ c)\ l$ holds for all option-lists $l \in O\,X$ by structural induction on l.

For the base case, suppose that $l = \varepsilon$. Then we have the following derivation by the definitions of *osel*, *map*, and *omap*.

$$\begin{aligned} (\mathit{osel}\ c \circ \mathit{omap}\ \varphi)\ \varepsilon &= \mathit{osel}\ c\ \varepsilon \\ &= \varepsilon \\ &= \mathit{map}\ \varphi\ \varepsilon \\ &= (\mathit{map}\ \varphi \circ \mathit{osel}\ c)\ \varepsilon \end{aligned}$$

For the induction step, suppose that $l = v :: l'$, for some variational option $v \in V\,X_\perp$ and option-list $l' \in O\,X$ with $(\mathit{osel}\ c \circ \mathit{omap}\ \varphi)\ l' = (\mathit{map}\ \varphi \circ \mathit{osel}\ c)\ l'$. Notice that $\mathit{sel}\ c\ v$ is either $\perp$ or some value $x \in X$.

If $\mathit{sel}\ c\ v = \perp$, then $\mathit{sel}\ c\ (\mathit{hmap}\ \varphi\ v) = \perp$ by Lemma 3.1 and we have the following derivation by the definitions of *osel* and *omap* and our inductive hypothesis.

$$\begin{aligned} (\mathit{osel}\ c \circ \mathit{omap}\ \varphi)\ (v :: l') &= \mathit{osel}\ c\ (\mathit{hmap}\ \varphi\ v :: \mathit{omap}\ \varphi\ l') \\ &= \mathit{osel}\ c\ (\mathit{omap}\ \varphi\ l') \\ &= (\mathit{map}\ \varphi \circ \mathit{osel}\ c)\ l' \\ &= (\mathit{map}\ \varphi \circ \mathit{osel}\ c)\ (v :: l') \end{aligned}$$

If $sel\ c\ v = x$ for some value $x \in X$, then $sel\ c\ (hmap\ \varphi\ v) = \varphi\ x$ by Lemma 3.2 and we have the following derivation by the definitions of *osel*, *map*, and *omap*, and our inductive hypothesis.

$$\begin{aligned}(osel\ c \circ omap\ \varphi)\ (v :: l') &= osel\ c\ (hmap\ \varphi\ v :: omap\ \varphi\ l') \\ &= \varphi\ x :: osel\ c\ (omap\ \varphi\ l') \\ &= \varphi\ x :: (map\ \varphi \circ osel\ c)\ l' \\ &= map\ \varphi\ (x :: osel\ c\ l') \\ &= (map\ \varphi \circ osel\ c)\ (v :: l')\end{aligned}$$

This completes the proof. ■

3.2.2 Bind Operation

The list type constructor $L\,(\cdot)$ together with some bind and unit operations form a monad on the category of types [Wadler, 1992, Wadler, Philip, 1992, Wadler, 1995]. The standard bind operation for lists is given by the function *bind*, which is defined for arbitrary types X and Y by the equations in Figure 3.8. This function applies a function to the elements of a list and joins the results. The infix binary operator $(+\!\!+)$ denotes a concatenation (or append) operation for lists. The bind operation for lists is sometimes referred to as a concatenation-map (or flat-map) operation in programming languages.

$$\begin{aligned}&bind : (X \to L\,Y) \to L\,X \to L\,Y \\ &bind\ \varphi\ \varepsilon \qquad\quad = \varepsilon \\ &bind\ \varphi\ (x :: l) = \varphi\ x +\!\!+ bind\ \varphi\ l\end{aligned}$$

Figure 3.8: Bind operation for lists.

It can be shown that the option-list type constructor $O\,(\cdot)$ together with some bind and unit operations form a monad on the category of types. The bind operation for option-lists is given by the function *obind*, which is defined for arbitrary types X and Y by the equations in Figure 3.9. This function applies a function to the elements of an option-list and joins the results.

$$
\begin{aligned}
&\mathit{obind} : (X \to O\,Y) \to O\,X \to O\,Y \\
&\mathit{obind}\ \varphi\ \varepsilon \qquad\quad = \varepsilon \\
&\mathit{obind}\ \varphi\ (v :: l) = \mathit{hbind}\ \varphi\ v +\!\!+ \mathit{obind}\ \varphi\ l
\end{aligned}
$$

Figure 3.9: Bind operation for option-lists.

The function *hbind* is defined for arbitrary types X and Y by the equations in Figure 3.10. This function applies a function to the variants of a variational option and joins the results.

$$
\begin{aligned}
&\mathit{hbind} : (X \to O\,Y) \to V\,X_\perp \to O\,Y \\
&\mathit{hbind}\ \varphi\ \perp \qquad\quad = \varepsilon \\
&\mathit{hbind}\ \varphi\ x \qquad\quad = \varphi\ x \\
&\mathit{hbind}\ \varphi\ f\langle v_1, v_2\rangle = \mathit{hzip}\ f\ (\mathit{hbind}\ \varphi\ v_1)\ (\mathit{hbind}\ \varphi\ v_2)
\end{aligned}
$$

Figure 3.10: *hbind* function definition.

The function *hzip* is defined for an arbitrary type X by the equations in Figure 3.11. This function forms an option-list where the elements are choices with a given formula as a label and elements of two given option-lists as alternatives.

$$
\begin{aligned}
&\mathit{hzip} : F \to O\,X \to O\,X \to O\,X \\
&\mathit{hzip}\ f\ \varepsilon\ l && = \mathit{map}\ (\lambda v.f\langle \bot, v\rangle)\ l \\
&\mathit{hzip}\ f\ l\ \varepsilon && = \mathit{map}\ (\lambda v.f\langle v, \bot\rangle)\ l \\
&\mathit{hzip}\ f\ (v_1 :: l_1)\ (v_2 :: l_2) && = f\langle v_1, v_2\rangle :: \mathit{hzip}\ f\ l_1\ l_2
\end{aligned}
$$

Figure 3.11: *hzip* function definition.

We use the definition of soundness (at a configuration) to define soundness of the bind operation for option-lists. The bind operation for option-lists is *sound* if for each configuration $c \in C$ and function $\varphi : X \to O\,Y$, the function *obind* φ is sound at c and with respect to the function *bind* ψ, where $\psi = \mathit{osel}\ c \circ \varphi$. This is stated formally by Theorem 3.4. We prove this theorem by structural induction on option-lists but the proof is rather tedious and so it is omitted here, see Appendix A.3.

Theorem 3.4. For all configurations $c \in C$ and functions $\varphi : X \to O\,X$, the equality $\mathit{osel}\ c \circ \mathit{obind}\ \varphi = \mathit{bind}\ \psi \circ \mathit{osel}\ c$ holds, where $\psi = \mathit{osel}\ c \circ \varphi$. Equivalently, the following diagram commutes:

$$
\begin{array}{ccc}
O\,X & \xrightarrow{\mathit{obind}\ \varphi} & O\,Y \\
{\scriptstyle \mathit{osel}\ c}\downarrow & & \downarrow{\scriptstyle \mathit{osel}\ c} \\
L\,X & \xrightarrow[\mathit{bind}\ \psi]{} & L\,Y
\end{array}
$$

Chapter 4 – Conclusion

We have presented semantic equivalence rules for an extension of the choice calculus and sound operations for an implementation of variational lists. The choice calculus is a calculus for describing variation and the formula choice calculus is an extension with formulas. We have proven semantic equivalence rules for the formula choice calculus. Variational lists are functional data structures for representing and computing with variation in lists using the choice calculus. We have proven map and bind operations are sound for an implementation of variational lists.

Appendix A – Formal Verification

In this appendix, we list verified proofs written in the language of the Coq proof assistant [Bertot and Castéran, 2004]. The properties from Chapter 2 are verified in Appendix A.1 and Appendix A.2 and the source code is available online.[1] The properties from Chapter 3 are verified in Appendix A.3 and the source code is also available online.[2]

A.1 Formulas

```
(** * Formula *)

Require Import Bool.
Require Import Relations.Relation_Definitions.
Require Import Classes.Morphisms.
Require Import Setoids.Setoid.

Module Formula.

(** ** Syntax *)
(** Syntax of formulas is Boolean expressions in dimensions and tags. *)

(** Dimensions and tags. *)
Definition dim := nat.
Definition tag := bool.
Definition L : tag := true.
Definition R : tag := false.

(** Formula syntax. *)
Inductive formula : Type :=
```

[1] https://github.com/hubbards/FCC-Coq
[2] https://github.com/hubbards/VP-Coq

```
  | litT : tag -> formula
  | litD : dim -> formula
  | comp : formula -> formula
  | join : formula -> formula -> formula
  | meet : formula -> formula -> formula.

Notation "~ f" := (comp f) (at level 75, right associativity).
Infix "\/" := join (at level 85, right associativity).
Infix "/\" := meet (at level 80, right associativity).

(** ** Semantics *)
(** The semantics of a formula is a function from configurations to tags. *)

(** Configurations. *)
Definition config := dim -> tag.

(** Formula semantics. *)
Fixpoint semF (f : formula) (c : config) : tag :=
  match f with
  | litT t   => t
  | litD d   => c d
  | ~ f      => negb (semF f c)
  | f1 \/ f2 => (semF f1 c) || (semF f2 c)
  | f1 /\ f2 => (semF f1 c) && (semF f2 c)
  end.

(** ** Semantic Equivalence Rules *)
(** Statement and proof of semantic equivalence rules for formulas from my
    thesis. Multiple proofs are given when it is instructive. *)

(** Semantic equivalence for formulas. *)
Definition equivF : relation formula :=
  fun f f' => forall c, (semF f c) = (semF f' c).

Infix "=f=" := equivF (at level 70) : type_scope.

(** Formula equivalence is reflexive. *)
```

```
Remark equivF_refl : Reflexive equivF.
Proof.
  intros x c.
  reflexivity.
Qed.

(** Formula equivalence is symmetric. *)
Remark equivF_sym : Symmetric equivF.
Proof.
  intros x y H c.
  symmetry.
  apply H.
Qed.

(** Formula equivalence is transitive. *)
Remark equivF_trans : Transitive equivF.
Proof.
  intros x y z H1 H2 c.
  transitivity (semF y c).
    apply H1.
    apply H2.
Qed.

(** Formula equivalence is an equivalence relation. *)
Instance eqF : Equivalence equivF.
Proof.
  split.
    apply equivF_refl.
    apply equivF_sym.
    apply equivF_trans.
Qed.

(** Congruence rule for complement. *)
Remark comp_cong : forall f f' : formula,
                   f =f= f' ->
                   (~ f) =f= (~ f').
Proof.
```

```
  intros f f' H c.
  simpl.
  rewrite -> H.
  reflexivity.
Qed.

(** Left congruence rule for join. *)
Remark join_cong_l : forall l l' r : formula,
                    l =f= l' ->
                    (l \/ r) =f= (l' \/ r).
Proof.
  intros l l' r H c.
  simpl.
  rewrite -> H.
  reflexivity.
Qed.

(** Right congruence rule for join. *)
Remark join_cong_r : forall l r r' : formula,
                    r =f= r' ->
                    (l \/ r) =f= (l \/ r').
Proof.
  intros l r r' H c.
  simpl.
  rewrite -> H.
  reflexivity.
Qed.

(** Congruence rule for join. *)
Remark join_cong : forall l l' r r' : formula,
                   l =f= l' -> r =f= r' ->
                   (l \/ r) =f= (l' \/ r').
Proof.
  intros l l' r r' Hl Hr.
  rewrite -> join_cong_l by apply Hl.
  rewrite -> join_cong_r by apply Hr.
  reflexivity.
```

```
Qed.

(** Left congruence rule for meet. *)
Remark meet_cong_l : forall l l' r : formula,
                    l =f= l' ->
                    (l /\ r) =f= (l' /\ r).
Proof.
  intros l l' r H c.
  simpl.
  rewrite -> H.
  reflexivity.
Qed.

(** Right congruence rule for meet. *)
Remark meet_cong_r : forall l r r' : formula,
                    r =f= r' ->
                    (l /\ r) =f= (l /\ r').
Proof.
  intros l r r' H c.
  simpl.
  rewrite -> H.
  reflexivity.
Qed.

(** Congruence rule for meet. *)
Remark meet_cong : forall l l' r r' : formula,
                   l =f= l' -> r =f= r' ->
                   (l /\ r) =f= (l' /\ r').
Proof.
  intros l l' r r' Hl Hr.
  rewrite -> meet_cong_l by apply Hl.
  rewrite -> meet_cong_r by apply Hr.
  reflexivity.
Qed.

(** Join is associative. *)
Theorem join_assoc : forall x y z : formula,
```

```
                    (x \/ y \/ z) =f= ((x \/ y) \/ z).
Proof.
  intros x y z c.
  apply orb_assoc.
Qed.

(** Meet is associative. *)
Theorem meet_assoc : forall x y z : formula,
                    (x /\ y /\ z) =f= ((x /\ y) /\ z).
Proof.
  intros x y z c.
  apply andb_assoc.
Qed.

(** Join is commutative. *)
Theorem join_comm : forall x y : formula,
                     (x \/ y) =f= (y \/ x).
Proof.
  intros x y c.
  apply orb_comm.
Qed.

(** Meet is commutative. *)
Theorem meet_comm : forall x y : formula,
                    (x /\ y) =f= (y /\ x).
Proof.
  intros x y c.
  apply andb_comm.
Qed.

(** Join distributes over meet. *)
Theorem join_meet_dist_r : forall x y z : formula,
                              (x \/ y /\ z) =f= ((x \/ y) /\ (x \/ z)).
Proof.
  intros x y z c.
  apply orb_andb_distrib_r.
Qed.
```

```
(** Meet distributes over join. *)
Theorem meet_join_dist_r : forall x y z : formula,
                             (x /\ (y \/ z)) =f= (x /\ y \/ x /\ z).
Proof.
  intros x y z c.
  apply andb_orb_distrib_r.
Qed.

(** Join is idempotent. *)
Theorem join_diag : forall f : formula,
                    (f \/ f) =f= f.
Proof.
  intros f c.
  apply orb_diag.
Qed.

(** Meet is idempotent. *)
Theorem meet_diag : forall f : formula,
                    (f /\ f) =f= f.
Proof.
  intros f c.
  apply andb_diag.
Qed.

(** De Morgan's law for join. *)
Theorem comp_join : forall x y : formula,
                    (~ (x \/ y)) =f= (~ x /\ ~ y).
Proof.
  intros x y c.
  apply negb_orb.
Qed.

(** De Morgan's law for meet. *)
Theorem comp_meet : forall x y : formula,
                    (~ (x /\ y)) =f= (~ x \/ ~ y).
Proof.
```

```
  intros x y c.
  apply negb_andb.
Qed.

(** Complementation for join. *)
Theorem join_comp_r : forall f : formula,
                      (f \/ ~ f) =f= litT L.
Proof.
  intros f c.
  apply orb_negb_r.
Qed.

(** Complementation for meet. *)
Theorem meet_comp_r : forall f : formula,
                      (f /\ ~ f) =f= litT R.
Proof.
  intros f c.
  apply andb_negb_r.
Qed.

(** Right is a left identity for join. *)
Theorem join_id_l : forall f : formula,
                    (litT R \/ f) =f= f.
Proof.
  intros f c.
  apply orb_false_l.
Qed.

(** Right is a right identity for join. *)
Theorem join_id_r : forall f : formula,
                    (f \/ litT R) =f= f.
Proof.
  intros f c.
  apply orb_false_r.
Qed.

(** Left is a left identity for meet. *)
```

```
Theorem meet_id_l : forall f : formula,
                    (litT L /\ f) =f= f.
Proof.
  intros f c.
  apply andb_true_l.
Qed.

(** Left is a right identity for meet. *)
Theorem meet_id_r : forall f : formula,
                    (f /\ litT L) =f= f.
Proof.
  intros f c.
  apply andb_true_r.
Qed.

(** Left is a left annihilator for join. *)
Theorem join_ann_l : forall f : formula,
                     (litT L \/ f) =f= litT L.
Proof.
  intros f c.
  apply orb_true_l.
Qed.

(** Left is a right annihilator for join. *)
Theorem join_ann_r : forall f : formula,
                     (f \/ litT L) =f= litT L.
Proof.
  intros f c.
  apply orb_true_r.
Qed.

(** Right is a left annihilator for meet. *)
Theorem meet_ann_l : forall f : formula,
                     (litT R /\ f) =f= litT R.
Proof.
  intros f c.
  apply andb_false_l.
```

```
Qed.

(** Right is a right annihilator for meet. *)
Theorem meet_ann_r : forall f : formula,
                     (f /\ litT R) =f= litT R.
Proof.
  intros f c.
  apply andb_false_r.
Qed.

(** Complement of left is right. *)
Theorem comp_l_r : (~ litT L) =f= litT R.
Proof.
  intro c.
  reflexivity.
Qed.

(** Complement of right is left. *)
Theorem comp_r_l : (~ litT R) =f= litT L.
Proof.
  intro c.
  reflexivity.
Qed.

(** Complement is an involution. *)
Theorem comp_invo : forall f : formula,
                    (~ ~ f) =f= f.
Proof.
  intros f c.
  apply negb_involutive.
Qed.

End Formula.
```

A.2 Formula Choice Calculus

```
(** * Formula Choice Calculus (FCC) *)

Require Import Bool.
Require Import Relations.Relation_Definitions.
Require Import Classes.Morphisms.
Require Import Setoids.Setoid.

Load Formula.
Import Formula.

Module FCC.

(** ** Syntax *)
(** Syntax of choice calculus expressions with global dimensions and formula
    choices. The object language is binary trees. *)

(** Object language syntax. *)
Inductive obj : Type :=
  | empty : obj
  | tree  : unit -> obj -> obj -> obj.

(** Expression syntax. *)
Inductive cc : Type :=
  | empty' : cc
  | tree'  : unit -> cc -> cc -> cc
  | chc    : formula -> cc -> cc -> cc.

(** ** Semantics *)
(** The semantics of a choice calculus expression is a function from
    configurations to terms in the object language, i.e., binary trees. *)

(** Expression semantics. *)
Fixpoint semE (e : cc) (c : config) : obj :=
  match e with
  | empty'      => empty
```

```
  | tree' x l r => tree x (semE l c) (semE r c)
  | chc f l r   => if semF f c then semE l c else semE r c
  end.

(** ** Semantic Equivalence Rules *)
(** Statement and proof of semantic equivalence rules for expressions from my
    thesis. Multiple proofs are given when it is instructive. *)

(** Semantic equivalence for expressions. *)
Definition equivE : relation cc :=
  fun e e' => forall c, (semE e c) = (semE e' c).

Infix "=e=" := equivE (at level 70) : type_scope.

(** Expression equivalence is reflexive. *)
Remark equivE_refl : Reflexive equivE.
Proof.
  intros x c.
  reflexivity.
Qed.

(** Expression equivalence is symmetric. *)
Remark equivE_sym : Symmetric equivE.
Proof.
  intros x y H c.
  symmetry.
  apply H.
Qed.

(** Expression equivalence is transitive. *)
Remark equivE_trans : Transitive equivE.
Proof.
  intros x y z H1 H2 c.
  transitivity (semE y c).
    apply H1.
    apply H2.
Qed.
```

```
(** Expression equivalence is an equivalence relation. *)
Instance eqE : Equivalence equivE.
Proof.
  split.
    apply equivE_refl.
    apply equivE_sym.
    apply equivE_trans.
Qed.

(** Choice transposition rule. *)
Theorem chc_trans : forall (f : formula) (l r : cc),
                    chc f l r =e= chc (~ f) r l.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f l r c.
  simpl.
  destruct (semF f c);
    reflexivity.
Qed.

(** AST-L-Congruence rule. *)
Remark ast_l_cong : forall l l' r : cc,
                    l =e= l' ->
                    tree' tt l r =e= tree' tt l' r.
Proof.
  intros l l' r H c.
  simpl.
  rewrite -> H.
  reflexivity.
Qed.

(** AST-R-Congruence rule. *)
Remark ast_r_cong : forall l r r' : cc,
                    r =e= r' ->
                    tree' tt l r =e= tree' tt l r'.
Proof.
```

```
  intros l r r' H c.
  simpl.
  rewrite -> H.
  reflexivity.
Qed.

(** Choice congruence rule for labels. *)
Remark chc_cong_f : forall (f f' : formula) (l r : cc),
                    f =f= f' ->
                    chc f l r =e= chc f' l r.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f f' l r H c.
  simpl.
  rewrite -> H.
  reflexivity.
Qed.

(** Choice congruence rule for left alternatives. *)
Remark chc_cong_l : forall (f : formula) (l l' r : cc),
                    l =e= l' ->
                    chc f l r =e= chc f l' r.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f l l' r H c.
  simpl.
  rewrite -> H.
  reflexivity.
Qed.

(** Choice congruence rule for right alternatives. *)
Remark chc_cong_r : forall (f : formula) (l r r' : cc),
                    r =e= r' ->
                    chc f l r =e= chc f l r'.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f l r r' H c.
```

```
  simpl.
  rewrite -> H.
  reflexivity.
Restart.
  (* Proof by deriving from [chc_cong_l]. *)
  intros f l r r' H.
  rewrite -> chc_trans.
  rewrite -> chc_cong_l by apply H.
  rewrite <- chc_trans.
  reflexivity.
Qed.

(** Choice idempotence rule. *)
Theorem chc_idemp : forall (f : formula) (e : cc),
                    chc f e e =e= e.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f e c.
  simpl.
  destruct (semF f c);
    reflexivity.
Qed.

(** Choice simplification rule for left label. *)
Theorem chc_f_l : forall (l r : cc),
                  chc (litT L) l r =e= l.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros l r c.
  reflexivity.
Qed.

(** Choice simplification rule for right label. *)
Theorem chc_f_r : forall (l r : cc),
                  chc (litT R) l r =e= r.
Proof.
  (* Proof by unfolding [equivE]. *)
```

```
  intros l r c.
  reflexivity.
Restart.
  (* Proof by deriving from [chc_f_l]. *)
  intros l r.
  rewrite -> chc_trans.
  rewrite -> chc_cong_f by apply comp_r_l.
  apply chc_f_l.
Qed.

(** Choice label join rule. *)
Theorem chc_f_join : forall (f1 f2 : formula) (l r : cc),
                     chc f1 l (chc f2 l r) =e= chc (f1 \/ f2) l r.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f1 f2 l r c.
  simpl.
  destruct (semF f1 c);
    reflexivity.
Qed.

(** Choice label meet rule. *)
Theorem chc_f_meet : forall (f1 f2 : formula) (l r : cc),
                     chc f1 (chc f2 l r) r =e= chc (f1 /\ f2) l r.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f1 f2 l r c.
  simpl.
  destruct (semF f1 c);
    reflexivity.
Restart.
  (* Proof by deriving from [chc_f_join]. *)
  intros f1 f2 l r.
  rewrite -> chc_cong_l with (l' := chc (~ f2) r l) by apply chc_trans.
  rewrite -> chc_trans.
  rewrite -> chc_f_join.
  rewrite -> chc_cong_f with (f' := ~ (f1 /\ f2)).
```

```
  rewrite <- chc_trans.
  reflexivity.
  symmetry.
  apply comp_meet.
Qed.

(** Choice label join complement rule. *)
Theorem chc_f_join_comp : forall (f1 f2 : formula) (l r : cc),
                          chc f1 l (chc f2 r l) =e= chc (f1 \/ ~ f2) l r.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f1 f2 l r c.
  simpl.
  destruct (semF f1 c);
    simpl;
    try rewrite -> negb_if;
    reflexivity.
Restart.
  (* Proof by deriving from [chc_f_join]. *)
  intros f1 f2 l r.
  rewrite -> chc_cong_r with (r' := chc (~ f2) l r) by apply chc_trans.
  rewrite -> chc_f_join.
  reflexivity.
Qed.

(** Choice label meet complement rule. *)
Theorem chc_f_meet_comp : forall (f1 f2 : formula) (l r : cc),
                          chc f1 (chc f2 r l) r =e= chc (f1 /\ ~ f2) l r.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f1 f2 l r c.
  simpl.
  destruct (semF f1 c);
    simpl;
    try rewrite -> negb_if;
    reflexivity.
Restart.
```

```
  (* Proof by deriving from [chc_f_meet]. *)
  intros f1 f2 l r.
  rewrite -> chc_cong_l with (l' := chc (~ f2) l r) by apply chc_trans.
  rewrite -> chc_f_meet.
  reflexivity.
Qed.

(** C-C-Merge rule. *)
Theorem cc_merge : forall (f : formula) (l r e e' : cc),
                   chc f (chc f l e) (chc f e' r) =e= chc f l r.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f l r e e' c.
  simpl.
  destruct (semF f c);
    reflexivity.
Qed.

(** C-C-Merge rule for the case where the nested choice appears in the left
    alternative. *)
Theorem cc_merge_l : forall (f : formula) (l r e : cc),
                     chc f (chc f l e) r =e= chc f l r.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f l r e c.
  simpl.
  destruct (semF f c);
    reflexivity.
Restart.
  (* Proof by deriving from [cc_merge]. *)
  intros f l r e.
  rewrite <- chc_cong_r with (r := chc f r r) by apply chc_idemp.
  rewrite -> cc_merge.
  reflexivity.
Qed.

(** C-C-Merge rule for the case where the nested choice appears in the right
```

```
    alternative. *)
Theorem cc_merge_r : forall (f : formula) (l r e : cc),
                     chc f l (chc f e r) =e= chc f l r.
Proof.
  (* Proof by deriving from [cc_merge_l]. *)
  intros f l r e.
  rewrite -> chc_cong_r with (r' := chc (~ f) r e) by apply chc_trans.
  rewrite -> chc_trans.
  rewrite -> cc_merge_l.
  rewrite <- chc_trans.
  reflexivity.
Qed.

(** AST-Factoring rule. *)
Theorem ast_factor : forall (f : formula) (l l' r r' : cc),
                     chc f (tree' tt l r) (tree' tt l' r') =e=
                     tree' tt (chc f l l') (chc f r r').
Proof.
  intros f l l' r r' c.
  simpl.
  destruct (semF f c);
    reflexivity.
Qed.

(** C-C-Swap rule. *)
Theorem cc_swap : forall (f1 f2 : formula) (e1 e2 e3 e4 : cc),
                  chc f1 (chc f2 e1 e2) (chc f2 e3 e4) =e=
                  chc f2 (chc f1 e1 e3) (chc f1 e2 e4).
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f1 f2 e1 e2 e3 e4 c.
  simpl.
  destruct (semF f1 c);
    reflexivity.
Qed.

(** C-C-Swap rule for the case where the nested choice appears in the left
```

```
    alternative of the simpler form. *)
Theorem cc_swap_l : forall (f f' : formula) (l r r' : cc),
                    chc f' (chc f l r') (chc f r r') =e=
                    chc f (chc f' l r) r'.
Proof.
  (* Proof by unfolding [equivE]. *)
  intros f f' l r r' c.
  simpl.
  destruct (semF f' c);
    reflexivity.
Restart.
  (* Proof by deriving from [cc_swap]. *)
  intros f f' l r r'.
  rewrite -> cc_swap.
  rewrite -> chc_cong_r by apply chc_idemp.
  reflexivity.
Qed.

(** C-C-Swap rule for the case where the nested choice appears in the right
    alternative of the simpler form. *)
Theorem cc_swap_r : forall (f f' : formula) (l l' r : cc),
                    chc f' (chc f l l') (chc f l r) =e=
                    chc f l (chc f' l' r).
Proof.
  (* Proof by deriving from [cc_swap_l]. *)
  intros f f' l l' r.
  rewrite -> chc_cong_l with (l' := chc (~ f) l' l) by apply chc_trans.
  rewrite -> chc_cong_r with (r' := chc (~ f) r l) by apply chc_trans.
  rewrite -> cc_swap_l.
  rewrite <- chc_trans.
  reflexivity.
Qed.

(** ** Examples *)
(** Examples of some additional properties and derivations by semantic
    equivalence rules. *)
Module Examples.
```

```
(** Flip operation. *)
Fixpoint flip (e : cc) : cc :=
  match e with
  | chc f l r => chc (~ f) (flip r) (flip l)
  | _         => e
  end.

(** The flip operation is an involution. *)
Example flip_invo : forall e : cc,
                    flip (flip e) =e= e.
Proof.
  induction e as [n | n l IHl r IHr | f l IHl r IHr].
  (* Case: [e = leaf' n]. *)
    reflexivity.
  (* Case: [e = node' n l r]. *)
    reflexivity.
  (* Case: [e = chc f l r]. *)
    simpl.
    rewrite -> chc_cong_f by apply comp_invo.
    rewrite -> chc_cong_l by apply IHl.
    rewrite -> chc_cong_r by apply IHr.
    reflexivity.
Qed.

End Examples.

End FCC.
```

A.3 Variational Programming

For simplicity, we use a non-generic map operation. However, this is not a fundamental restriction. We could use a generic map operation and the following proofs would not change since the logic does not depend on the list type parameter.

```
(** * Option-List (OList) *)

Require Import Bool.
Require Import List.
Import ListNotations.
Require Import Basics.

Module OList.

(** Dimensions and configurations. *)
Definition dim := nat.
Definition tag := bool.
Definition L : tag := true.
Definition R : tag := false.
Definition config := dim -> tag.

(** Formula syntax. *)
Inductive formula : Type :=
  | litT : tag -> formula
  | litD : dim -> formula
  | comp : formula -> formula
  | join : formula -> formula -> formula
  | meet : formula -> formula -> formula.

Notation "~ x" := (comp x) (at level 75, right associativity).
Infix "\/" := join (at level 85, right associativity).
Infix "/\" := meet (at level 80, right associativity).

(** Formula semantics. *)
Fixpoint semf (x : formula) (c : config) : tag :=
  match x with
  | litT t   => t
```

```
  | litD d   => c d
  | ~ x      => negb (semf x c)
  | x1 \/ x2 => (semf x1 c) || (semf x2 c)
  | x1 /\ x2 => (semf x1 c) && (semf x2 c)
  end.

(** Variational option definition. *)
Inductive var : Type :=
  | one : option nat -> var
  | chc : formula -> var -> var -> var.

(** Selection operation for variational option. *)
Fixpoint vsel (c : config) (v : var) : option nat :=
  match v with
  | one o      => o
  | chc x l r  => if semf x c then vsel c l else vsel c r
  end.

(** Option-list definition. *)
Definition olist := list var.

(** Selection operation for option-list. *)
Fixpoint osel (c : config) (o : olist) : list nat :=
  match o with
  | []     => []
  | h :: t => match vsel c h with
              | None   => osel c t
              | Some y => y :: osel c t
              end
  end.

(** ** Map *)
(** Definition of map operation for option-list and a formal proof of
    soundness. *)
Section Map.

(** Helper function of map operation for option-list. *)
```

```
Fixpoint hmap (f : nat -> nat) (v : var) : var :=
  match v with
  | one None     => one None
  | one (Some n) => one (Some (f n))
  | chc x l r    => chc x (hmap f l) (hmap f r)
  end.

(** Map operation for option-list. *)
Fixpoint omap (f : nat -> nat) (o : olist) : olist :=
  match o with
  | []     => []
  | h :: t => hmap f h :: omap f t
  end.

(* Lemma used in proof of [omap_sound]. *)
Lemma vsel_none : forall (c : config) (f : nat -> nat) (e : var),
                  vsel c e = None ->
                  vsel c (hmap f e) = None.
Proof.
  intros c f e H.
  induction e as [o | x l IHl r IHr].
  (* Case: [e = one o]. *)
    destruct o as [n |].
    (* Subcase: [o = Some n]. *)
      simpl vsel in H.
      inversion H.
    (* Subcase: [o = None]. *)
      reflexivity.
  (* Case: [e = chc x l r]. *)
    simpl.
    simpl in H.
    destruct (semf x c).
    (* Subcase: [semf x c = L]. *)
      rewrite -> IHl by apply H.
      reflexivity.
    (* Subcase: [semf x c = R]. *)
      rewrite -> IHr by apply H.
```

```
      reflexivity.
Qed.

(* Lemma used in proof of [omap_sound]. *)
Lemma vsel_some : forall (c : config) (f : nat -> nat) (e : var) (n : nat),
                  vsel c e = Some n ->
                  vsel c (hmap f e) = Some (f n).
Proof.
  intros c f e n H.
  induction e as [o | x l IHl r IHr].
  (* Case: [e = one o]. *)
    destruct o as [n' |].
    (* Subcase: [o = Some n']. *)
      simpl vsel in H.
      inversion H.
      reflexivity.
    (* Subcase: [o = None]. *)
      simpl vsel in H.
      inversion H.
  (* Case: [e = chc x l r]. *)
    simpl.
    simpl in H.
    destruct (semf x c).
    (* Subcase: [semf x c = L]. *)
      rewrite -> IHl by apply H.
      reflexivity.
    (* Subcase: [semf x c = R]. *)
      rewrite -> IHr by apply H.
      reflexivity.
Qed.

Infix "*" := compose.

(** The map operation for option-list is sound. *)
Theorem omap_sound : forall (c : config) (f : nat -> nat) (o : olist),
                       (osel c * omap f) o = (map f * osel c) o.
Proof.
```

```
  intros c f o.
  unfold compose.
  induction o as [| h t IH].
  (* Case: [o = nil]. *)
    reflexivity.
  (* Case: [o = cons h t]. *)
    simpl omap.
    destruct (vsel c h) as [n |] eqn : H.
    (* Subcase: [vsel c h = Some n]. *)
      simpl osel.
      rewrite -> vsel_some with (n := n) by apply H.
      rewrite -> H.
      simpl map.
      rewrite -> IH.
      reflexivity.
    (* Subcase: [vsel c h = None]. *)
      simpl osel.
      rewrite -> vsel_none by apply H.
      rewrite -> H.
      apply IH.
Qed.

End Map.

(** ** Bind *)
(** Definition of bind operation for option-list and a formal proof of
    soundness. *)
Section Bind.

(** Helper function of bind operation for option-list. *)
Fixpoint hzip (x : formula) (o o' : olist) : olist :=
  match o, o' with
  | [], _            => map (fun v : var => chc x (one None) v) o'
  | _, []            => map (fun v : var => chc x v (one None)) o
  | h :: t, h' :: t' => chc x h h' :: hzip x t t'
  end.
```

```
(** Helper function of bind operation for option-list. *)
Fixpoint hbind (f : nat -> olist) (v : var) : olist :=
  match v with
  | one None     => []
  | one (Some n) => f n
  | chc x l r    => hzip x (hbind f l) (hbind f r)
  end.

(** Bind operation for option-list. *)
Fixpoint obind (f : nat -> olist) (o : olist) : olist :=
  match o with
  | []     => []
  | h :: t => hbind f h ++ obind f t
  end.

(** Lemma used to simplify goals in later proofs. *)
Lemma osel_cons_chc_l :
  forall (c : config) (x : formula) (l r : var) (o : olist),
  semf x c = L ->
  osel c (chc x l r :: o) = osel c (l :: o).
Proof.
  intros c x l r o H.
  simpl.
  rewrite -> H.
  reflexivity.
Qed.

(** Lemma used to simplify goals in later proofs. *)
Lemma osel_cons_chc_r :
  forall (c : config) (x : formula) (l r : var) (o : olist),
  semf x c = R ->
  osel c (chc x l r :: o) = osel c (r :: o).
Proof.
  intros c x l r o H.
  simpl.
  rewrite -> H.
  reflexivity.
```

```
Qed.

(** Selection for option-list distributes over append. *)
Lemma osel_app : forall (c : config) (o o' : olist),
                 osel c (o ++ o') = osel c o ++ osel c o'.
Proof.
  intros c o o'.
  induction o as [| h t IH].
  (* Case: [o = nil]. *)
    reflexivity.
  (* Case: [o = cons h t]. *)
    simpl.
    destruct (vsel c h) as [n |].
    (* Subcase: [vsel c h = Some n]. *)
      rewrite <- app_comm_cons.
      rewrite -> IH.
      reflexivity.
    (* Subcase: [vsel c h = None]. *)
      apply IH.
Qed.

(** Lemma used to simplify goals in later proofs. *)
Lemma ozip_nil_l : forall (x : formula) (o : olist),
                   hzip x [] o = map (fun v : var => chc x (one None) v) o.
Proof.
  destruct o;
    unfold hzip;
    reflexivity.
Qed.

(** Lemma used to simplify goals in later proofs. *)
Lemma ozip_nil_r : forall (x : formula) (o : olist),
                   hzip x o [] = map (fun v : var => chc x v (one None)) o.
Proof.
  destruct o;
    unfold hzip;
    reflexivity.
```

```
Qed.

(** Lemma used to simplify goals in later proofs. *)
Lemma osel_ozip_l : forall (c : config) (x : formula),
                    semf x c = L ->
                    forall o o' : olist, osel c (hzip x o o') = osel c o.
Proof.
  intros c x H.
  assert (H' : forall o : olist, osel c (hzip x [] o) = []).
  (* Proof of assertion [H']. *)
    intro o.
    induction o as [| h t IH].
    (* Case: [o = nil]. *)
      reflexivity.
    (* Case: [o = cons h t]. *)
      rewrite -> ozip_nil_l.
      simpl map.
      rewrite -> osel_cons_chc_l by apply H.
      simpl.
      rewrite <- ozip_nil_l.
      apply IH.
  (* Proof of [osel_ozip_l]. *)
  intro o.
  induction o as [| h t IH].
  (* Case: [o = nil]. *)
    apply H'.
  (* Case: [o = cons h t]. *)
    destruct o' as [| h' t'];
      simpl hzip;
      rewrite -> osel_cons_chc_l by apply H;
      try (rewrite <- ozip_nil_r);
      destruct h as [o | y l r];
        try (destruct o);
        simpl;
        rewrite -> IH;
        reflexivity.
Qed.
```

```
(** Lemma used to simplify goals in later proofs. *)
Lemma osel_ozip_r : forall (c : config) (x : formula),
                    semf x c = R ->
                    forall o o' : olist,
                    osel c (hzip x o' o) = (osel c o).
Proof.
  intros c x H.
  assert (H' : forall o : olist, osel c (hzip x o []) = []).
  (* Proof of assertion [H']. *)
    intro o.
    induction o as [| h t IH].
    (* Case: [o = nil]. *)
      reflexivity.
    (* Case: [o = cons h t]. *)
      rewrite -> ozip_nil_r.
      simpl map.
      rewrite -> osel_cons_chc_r by apply H.
      simpl.
      rewrite <- ozip_nil_r.
      apply IH.
  (* Proof of [osel_ozip_r]. *)
  intros o.
  induction o as [| h t IH].
  (* Case: [o = nil]. *)
    apply H'.
  (* Case: [o = cons h t]. *)
    destruct o' as [| h' t'];
      simpl hzip;
      rewrite -> osel_cons_chc_r by apply H;
      try (rewrite <- ozip_nil_l);
      destruct h as [o | y l r];
        try (destruct o);
        simpl;
        try (rewrite <- ozip_nil_l);
        rewrite -> IH;
        reflexivity.
```

```
Qed.

Infix "∗" := compose.

(** The bind operation for option-list is sound. ∗)
Theorem obind_sound : forall (c : config) (f : nat -> olist) (o : olist),
                        (osel c ∗ obind f) o =
                        (flat_map (osel c ∗ f) ∗ osel c) o.
Proof.
  intros c f o.
  unfold compose.
  induction o as [| h t IH].
  (∗ Case: [o = nil]. ∗)
    reflexivity.
  (∗ Case: [o = cons h t]. ∗)
    simpl obind.
    rewrite -> osel_app.
    induction h as [o | x l IHl r IHr].
    destruct o.
    (∗ Subcase: [h = one (Some n)]. ∗)
      simpl.
      rewrite -> IH.
      reflexivity.
    (∗ Subcase: [h = one None]. ∗)
      simpl.
      apply IH.
    (∗ Subcase: [h = chc x l r]. ∗)
      simpl hbind.
      destruct (semf x c) eqn : H.
        rewrite -> osel_ozip_l by apply H.
        rewrite -> osel_cons_chc_l by apply H.
        apply IHl.
        rewrite -> osel_ozip_r by apply H.
        rewrite -> osel_cons_chc_r by apply H.
        apply IHr.
Qed.
```

```
End Bind.

End OList.
```

Bibliography

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, New York, 2004.

E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba. SPLlift - Statically Analyzing Software Product Lines in Minutes Instead of Years. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2013.

D. S. Dummit and R. M. Foote. *Abstract Algebra*. John Wiley and Sons, third edition, 2004.

M. Erwig and E. Walkingshaw. Semantics First! Rethinking the Language Design Process. In *Int. Conf. on Software Language Engineering (SLE)*, volume 6940 of *LNCS*, pages 243–262, 2011a.

M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27, 2011b.

C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 311–320, 2008.

C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 805–824, 2011.

C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-based Product Lines. *ACM Trans. on Software Engineering and Methodology*, 21(3): 14:1–14:39, July 2012.

A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #ifdef Variability in C. In *Int. Workshop on Feature-Oriented Software Development*, pages 25–32, 2010.

P. Wadler. The essence of functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, 1992.

P. Wadler. Monads for Functional Programming. In *Advanced Functional Programing*, LNCS 925, pages 24–52, 1995.

Wadler, Philip. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

E. Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, 2013. http://hdl.handle.net/1957/40652.

E. Walkingshaw and M. Erwig. A Calculus for Modeling and Implementing Variation. In *ACM SIGPLAN Int. Conf. on Generative Programming and Component Engineering (GPCE)*, pages 132–140, 2012.

E. Walkingshaw and K. Ostermann. Projectional Editing of Variational Software. In *ACM SIGPLAN Int. Conf. on Generative Programming and Component Engineering (GPCE)*, pages 29–38, 2014.

E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *ACM SIGPLAN Symp. on New Ideas in Programming and Reflections on Software (Onward!)*, pages 213–226, 2014.