Report Number: WUCSE-2005-15

2005-04-15

What A Mesh: Dependent Data Types for Correct Mesh Manipulation Algorithms

Joel R. Brandt

The Edinburgh Logical Framework (LF) has been proposed as a system for expressing inductively defined sets. I will present an inductive definition of the set of manifold meshes in LF. This definition takes into account the topological characteri-zation of meshes, namely their Euler Characteristic. I will then present a set of dependent data types based on this inductive def-inition. These data types are defined in a programming language based on LF. The language's type checking guarantees that any typeable expression represents a correct manifold mesh. Furthermore, any mesh can be represented using these data types. Hence, the encoding is... **Read complete abstract on page 2.**

What A Mesh: Dependent Data Types for Correct Mesh Manipulation Algorithms

Joel R. Brandt

Complete Abstract:

The Edinburgh Logical Framework (LF) has been proposed as a system for expressing inductively defined sets. I will present an inductive definition of the set of manifold meshes in LF. This definition takes into account the topological characteri-zation of meshes, namely their Euler Characteristic. I will then present a set of dependent data types based on this inductive def-inition. These data types are defined in a programming language based on LF. The language's type checking guarantees that any typeable expression represents a correct manifold mesh. Furthermore, any mesh can be represented using these data types. Hence, the encoding is sound and complete.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2005-15

What A Mesh: Dependent Data Types for Correct Mesh Manipulation Algorithms, Master's Thesis, May 2005

Authors: Brandt, Joel R.

April 15, 2005

Abstract: The Edinburgh Logical Framework (LF) has been proposed as a system for expressing inductively defined sets. I will present an inductive definition of the set of manifold meshes in LF. This definition takes into account the topological characterization of meshes, namely their Euler Characteristic.

I will then present a set of dependent data types based on this inductive definition. These data types are defined in a programming language based on LF. The language's type checking guarantees that any typeable expression represents a correct manifold mesh. Furthermore, any mesh can be represented using these data types. Hence, the encoding is sound and complete.

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

WHAT A MESH: DEPENDENT DATA TYPES FOR CORRECT MESH MANIPULATION ALGORITHMS

by

Joel R. Brandt

Prepared under the direction of Aaron Stump and Cindy Grimm

Thesis presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment of the
requirements for the degree of

Master of Science

May 2005

Saint Louis, Missouri

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

STUDENT: Joel R. Brandt

ADVISORS: Aaron Stump and Cindy Grimm

May 2005
Saint Louis, Missouri

The Edinburgh Logical Framework (LF) has been proposed as a system for expressing inductively defined sets. I will present an inductive definition of the set of manifold meshes in LF. This definition takes into account the topological characterization of meshes, namely their Euler Characteristic.

I will then present a set of dependent data types based on this inductive definition. These data types are defined in a programming language based on LF. The language's type checking guarantees that any typeable expression represents a correct manifold mesh. Furthermore, any mesh can be represented using these data types. Hence, the encoding is sound and complete.

to Lindsay, Carol, Richard, Eric, Lisa, and Andrew

Contents

Figures

Acknowledgments

I would like to thank all of the excellent advisors I have had along the way, especially Kenneth Goldman, Aaron Stump, Cindy Grimm, M. Victor Wickerhauser, and William Smart. Additionally, I would like to thank the members of my research group, Ian Wehrman, Eddy Westbrook, Robert Klapper, and Li-Yang Tan, for all of their support and advice.

Joel R. Brandt

Washington University in Saint Louis
May 2005

Chapter 1

Introduction

Correct programming is difficult and verifying that a program is correct can be even more difficult. The reasons for this are numerous. One major contributing factor is that the proof of an algorithm's correctness is often not tied closely to the implementation of the algorithm.

Algorithms are typically written recursively. This necessitates a constructive proof of correctness if the programmer wishes to argue that the code is truly correct. However, in many domains, constructive proofs can be large and cumbersome. Mesh manipulation algorithms for computer graphics is one such domain.

Constructive proofs of topological properties are often difficult because many properties must be verified universally. For example, a mesh must be *orientable*, meaning intuitively that, at all points, a direction of "up" can be agreed upon [7]. (A Möbius strip is one common example of a non-orientable surface.) It is clear that a constructive proof which verifies this at every point in the mesh will be quite large.

As we shall see, writing constructive proofs by hand for all but the simplest of meshes (and perhaps even for the simplest of meshes) is a practically impossible task. Our goal, instead, is to allow the programmer to tie the construction of these proofs directly to the implementation of the algorithm. If an algorithm is written in such a way that it will produce not only the result, but also a proof that the result is correct, then one can be certain that the algorithm is correct.

We propose that these proofs be part of the type system, so that a mesh is correctly typed only if it contains a constructive proof that it is a valid mesh. Then, an algorithm will only type check if it produces this proof.

In this thesis, after some background information, we present our inductive definition of manifold meshes. We argue that this representation is faithful. Then, we present an embedding of these definitions in the Edinburgh Logical Framework and argue that our syntax is adequate. We encode this definition as data types in the programming language Rogue-Sigma-Pi which is based on the Edinburgh Logical Framework. We then conclude with a discussion of program verification through typing.

1.1 Major Contributions

This thesis makes the following contributions:

- Development of an inductive definition of manifold meshes.
- Embedding of manifold meshes in the Edinburgh Logical Framework
- Development of a set of dependent data types representing valid manifold meshes.
- Discussion of program verification through type systems, and thoughts on further language support for verification.

The remainder of this chapter provides background for the thesis. We first give an overview of the Edinburgh Logical Framework. We then discuss dependent types and the programming language Rogue-Sigma-Pi. Finally, we give an overview of manifold meshes.

1.2 Overview of the Edinburgh Logical Framework

We present only a brief introduction to the fragment of the Edinburgh Logical Framework (LF) that we will be using explicitly. The interested reader is referred to several other sources [10, 5].

kinds	K	$::=$	$\mathsf{type} \mid \Pi x\!:\!A.K$
families	A	$::=$	$a \mid \Pi x\!:\!A.B \mid AM$
objects	M	$::=$	$c \mid x \mid MN$

Figure 1-1: A portion of the LF grammar.

LF is a type theory based on a dependently-typed λ-calculus and was originally developed as a framework to define or represent logical systems. It has also served as the underlying type theory for programming languages like Twelf [11] and Rogue-Sigma-Pi [12]. Here, we will use it as a framework to represent inductively defined sets.

LF uses the notion of *judgments as types* [5]. This means that proof rules are expressed as typed terms, with the type of the term representing the judgment made. Then, proof checking amounts to type checking.

The syntax for LF is given by the grammar[1] for *kinds*, *families*, and *objects* as shown in Figure 1-1. In addition, LF contains *contexts* and *signatures*, both of which are finite sequences of definitions. A signature contains a list of types of kinds, families, and objects in the framework and a context contains types of bound variables. These two structures are used in type (proof) checking. Embedding a logic (or in our case, meshes) amounts to defining a signature for type checking. Then, any typeable term expressed using this signature will represent an object of that type.

For convenience, we will write $x : A \Rightarrow K$ and $x : A \Rightarrow B$ for $\Pi x : A.K$ and $\Pi x\!:\!A.B$ respectively. If x does not occur free in K or B, we may omit it. Successive occurrences of "$\Rightarrow$" associate to the right and capture-avoiding substitution occurs as expected.

The reader who is familiar with λ-calculus, but not with dependent types, may be somewhat confused by the expression $\Pi x\!:\!A.B$ given in the grammar for families. $\Pi x : A.B$ is simply the type of a function which takes in an object of type A and returns an object of type B, where B may be dependent on x. In the following section, we will see that this is analogous to the type of a type constructor in RSP. A type constructor takes in a term of some type and returns a new dependent type indexed by that term.

[1]The complete grammar for LF's syntax contains several other non-terminals for *kinds* and *families*, but we will not be using them here.

We do not present the typing rules for LF here, as they can be found elsewhere [5], and are beyond the scope of this thesis. They are, however, a (non-trivial) extension of the typing rules for typed λ-calculus.

1.3 Overview of Dependent Types

Dependent types appear in many modern programming languages [2, 15, 14]. Dependent types simply allow the type of a term to be indexed by a piece (or pieces) of data. For example, the type of a list could be indexed by a number indicating the length of the list. We write dependent types using a functional notation, so the type of our list would be $\mathsf{list}(n)$, where n is the length of the list.

Dependent types can be used for some level of static program verification. For example, an algorithm which operates on a list could take in a number n and a list of type $\mathsf{list}(n)$. The type checker would then guarantee that the list actually contained n elements.

1.3.1 The Syntax of Rogue-Sigma-Pi

Here we briefly introduce the syntax necessary for understanding the fragment of RSP used in this thesis. A more thorough discussion of RSP's syntax can be found elsewhere [14].

The "`::`" terminal is used infix to represent declarations. For example, the statement `num :: type;;` declares `num` to be a type within the system. Likewise, the "`:`" terminal is used for type ascription in the declaration of dependent types (as we will see later).

The "`=>`" terminal is a *representational arrow*. It is used infix to represent the types of curried functions. For example, `succ :: num => num;;` declares `succ` to be a term representing a function from `num`s to `num`s.

1.3.2 Dependent Types in Rogue-Sigma-Pi

Rogue-Sigma-Pi (RSP) [12, 14] is a dependently-typed programming language based on LF which we will use to encode our inductive definition of meshes. In RSP,

```
# numbers
num ::  type;;
zero ::  num;;
succ ::  num => num;;

# list type constructor
list ::  num => type;;

# empty list
nil ::  list zero;;

# cons function operating on lists
cons ::  t:trm => n:num => l:list n => list (succ n);;
```

Figure 1-2: Example RSP code showing dependently-typed lists.

dependent types are created with *type constructors*. A type constructor for the list example presented above can be seen in Figure 1-2.

In the example presented in Figure 1-2, we also introduce a non-dependent type num. We then create a term `zero` of type num, as well as a successor function which takes in nums and returns nums.

1.3.3 Using Dependent Types to Verify Data

In this thesis, our main use of dependent types is to statically verify properties about data. Continuing our example, we first introduce a `nil` term which represents an empty list. Naturally, it has type list(`zero`). We then define the CONS function on lists. This function first takes in the term to cons onto the list, then n of type number, and finally a list of type list(n) (all functions in RSP are represented in curried form). The resulting term has type list(SUCC(n)).

Because `nil` and terms created with CONS are the only terms which have type list, we have statically guaranteed that any typeable term of type list(n) will have n elements. Figure 1-3 shows examples of both typeable and untypeable list terms. The error in the second term would be caught by the type checker.

By indexing terms with more elaborate data, we can statically guarantee many more properties of our data. We will revisit this idea in Chapter 4.

```
# a typeable term of type list(succ(zero))
cons y (succ zero) (cons x zero nil)

# an untypeable term
cons y (succ (succ zero)) (cons x zero nil)
```

Figure 1-3: Example of an untypeable expression in RSP.

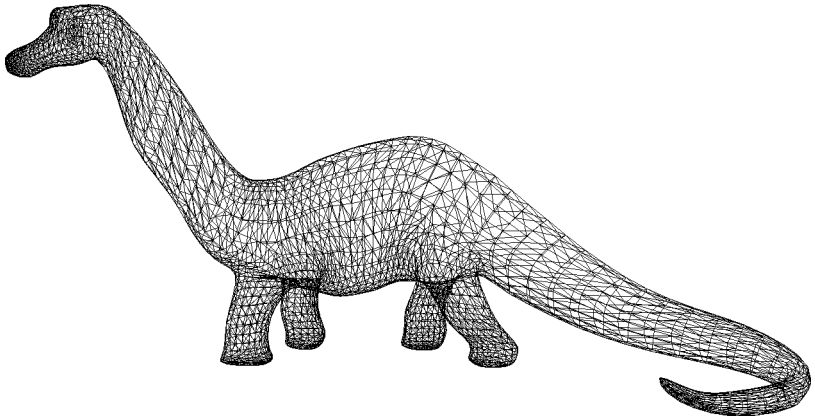

Figure 1-4: Example of a manifold mesh

1.4 Overview of Meshes

A manifold mesh consists of vertexes, directed edges, and faces, that together form a two-dimensional *connected*, *orientable*, *compact manifold*. Intuitively, a mesh can be thought of as a discrete representation of the surface of a 3-dimensional object. Indeed, Figure 1-4 gives an example of a manifold mesh representing the surface of a dinosaur.

Meshes are typically expressed as a series of complexes. The $\mathcal{C}_0$ complex is a set of vertexes. The $\mathcal{C}_1$ complex is a set of directed edges built from vertexes in the $\mathcal{C}_0$ complex. Finally, the $\mathcal{C}_2$ complex is a set of faces built from edges in the $\mathcal{C}_1$ complex. Because any polygon can be triangulated, we consider only triangular faces throughout this thesis.

Naturally, for a set of complexes to form a valid manifold mesh, certain properties must hold within and between the sets. In the following section, we will examine

each of the topological requirements of a mesh, and give constraints on the complexes which guarantee these requirements.

1.4.1 Properties of Manifold Meshes

In order for the complexes to represent a mesh, we must ensure that the structure is connected, orientable, and manifold. With these three properties, compactness is guaranteed, since each face is compact, and there are a finite number of them. We examine each characteristic separately.

Connectedness

Connectedness is the most straightforward property to guarantee. We simply require that the $\mathcal{C}_0$ and $\mathcal{C}_1$ complexes form a connected graph.

Orientability

Because the edges are directed, we can use them to define an orientation of each face. As such, we have our first constraint. The set of three edges which form each face must form a cycle. That is, the head of each edge in the face must be the tail of the edge following it.

Then, as we travel from face to face, we need the property that our orientation remain consistent. So, if two faces share an undirected edge, the directed edges in each of the faces must go in opposite directions. (The converse also holds; if two faces share the same directed edge, it is certain that the surface is not orientable.) From this, we get our second constraint. No directed edge can appear in more than one face.

Figure 1-5 shows several examples of face pairings. Note that the face pairing on the left has a consistent orientation, where as the one in the middle does not.

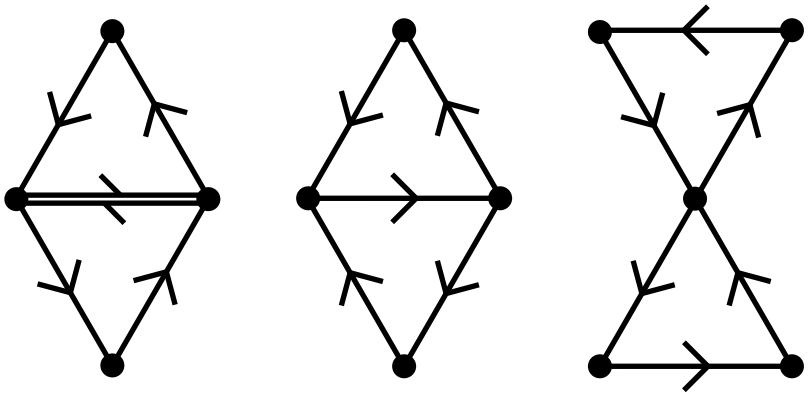

Figure 1-5: Examples of one well-formed face pairing (left) and two ill-formed face pairings (middle, right). The middle face pairing is not orientable, and the right face pairing is not manifold.

Manifoldness

A topological surface is manifold if each point on the surface is locally homeomorphic[2] to the plane, and each point on the boundary is locally homeomorphic to the half-plane. Any point inside a face certainly satisfies this property, as does any point in the middle of an undirected edge, whether it be an interior or boundary edge. (Note that a boundary edge is any directed edge where the opposite edge is not part of the mesh.)

Vertexes which are not part of any boundary edge will also certainly satisfy the property. However, vertexes which are part of a boundary edge may cause problems. If a *butterfly vertex* exists, as shown on the right in Figure 1-5, the manifold property will not hold. More formally, every boundary of the mesh must be a cycle made up of a disjoint set of edges and vertexes, and no boundary may intersect itself.

1.4.2 Topological Characterization

The topological characterization (or topology) of a mesh is often of interest. This is especially true when attempting to verify mesh manipulation algorithms. Often, one important part of correctness is ensuring that in manipulating the mesh the topology does not change.

[2]Two surfaces are *locally homeomorphic* at some point iff for some disc of radius $\epsilon > 0$ around the point, the two surfaces are identical.

The topology of a compact, connected surface can be completely expressed by its Euler Characteristic [7]. The Euler Characteristic is defined as

$$\chi(M) = |V| - |E| + |F|$$

where V is the set of vertexes, E is the set of *undirected* edges, and F is the set of faces.

Because of the importance of the topological characterization of a mesh, we will take great care to ensure that the Euler Characteristic is represented in our embedding.

Chapter 2

Inductive Definition of Manifold Meshes

In this chapter we present an inductive definition of manifold meshes. This inductive definition is based on the mesh properties expressed in Section 1.4. This definition is sound and complete, so we say that it is a *faithful* representation.

2.1 Overview of Notation

Our rules for the inductive definition of meshes are presented in deduction notation. The items above the line in a rule can be used to build the items below the line. In addition, some of these rules require that certain constraints are met. These constraints are listed below the rules.

The symbols v, e, f, and m (possibly with subscripts) represent vertexes, edges, faces and meshes, respectively, and the symbols V, E, and F represent sets of the first three.

Finally, we introduce two binary relations, "$\not\triangleleft$" and "$\sqsubset$", used infix. An edge e and a face set F are related by "$\not\triangleleft$" iff F does not contain any face using edge e. This is expressed more formally as $\forall f \in F.e \notin \partial(f)$, where $\partial(f)$ is the set of edges in f (i.e. the *boundary* of f).

Two sets S_1 and S_2 are related by "$\sqsubset$" iff every element in S_1 can be found in some element of S_2. For example, $V \sqsubset E$ if, for every vertex v in V, there

is some edge using vertex v in E. Again, this can be expressed more formally as $\forall s \in S_1.\exists t \in S_2.s \in \partial(t)$.

2.2 Mesh Primitives

Figure 2-1 presents the rules for the construction of mesh primitives of vertexes, edges, and faces.

$$\frac{}{v}\ \textsf{make-vertex}$$

$$\frac{v_1 \quad v_2}{e(v_1, v_2)}\ \textsf{make-edge}^*$$

$$\frac{e_1(v_1, v_2) \quad e_2(v_2, v_3) \quad e_3(v_3, v_1)}{f(e_1, e_2, e_3)}\ \textsf{make-face}$$

* if $v_1 \neq v_2$

Figure 2-1: Rules for mesh primitives

In the rule make-edge, we have our first constraint, namely, that an edge be constructed from two unique vertexes. As we build up more rules, these constraints will become more complex in order to meet the topological requirements expressed in Section 1.4. Our proofs of soundness and completeness will draw on these constraints.

Note that there is no need to explicitly enforce that the three edges used to make a face form a simple cycle. Because the two vertexes from each edge are guaranteed to be unique, we are guaranteed a cycle.

2.3 The Complexes

The $\mathcal{C}_0$, $\mathcal{C}_1$, and $\mathcal{C}_2$ complexes are defined very similarly. For each, there is a *nil* constructor rule that allows the construction of an empty complex. Because higher

complexes must be built out of units from lower complexes, we require the lower complexes in the antecedent.

2.3.1 The $\mathcal{C}_0$ Complex

Figure 2-2 presents the rules for the construction of the $\mathcal{C}_0$ complex.

$$\frac{}{\mathcal{C}_0(\emptyset)}\ \textsf{c0-nil}$$

$$\frac{\mathcal{C}_0(V)\quad v}{\mathcal{C}_0(V \cup \{v\})}\ \textsf{c0-cons}^*$$

* if $v \notin V$

Figure 2-2: Rules for the $\mathcal{C}_0$ complex

There are no constraints on what vertexes can be placed into the $\mathcal{C}_0$ complex (except that a new vertex cannot already be a member of the set). The following two complexes will place tighter constraints on what can be added.

2.3.2 The $\mathcal{C}_1$ Complex

Figure 2-3 presents the rules for the construction of the $\mathcal{C}_1$ complex.

In our definition, the $\mathcal{C}_1$ complex is made up of *two* sets of vertexes and a set of edges. The first vertex set is the vertex set from the $\mathcal{C}_0$ complex on top of which this complex will be built. This is evident from the c1-nil rule.

In order to understand the use of the second vertex set, we must first observe that there are two rules that add edges to the complex. One rule allows the addition of inner edges. When this rule is used, both directed edges are added to the edge set. The other rule allows the addition of boundary edges. This rule has the additional constraint that the first vertex of the edge not be in the second vertex set. Upon adding the edge, this vertex is added to the set. We will argue later that this is necessary to guarantee manifoldness.

$$\frac{\mathcal{C}_0(V)}{\mathcal{C}_1(V,\emptyset,\emptyset)}\ \textsf{c1-nil}$$

$$\frac{\mathcal{C}_1(V,V_b,E)\quad e(v_1,v_2)}{\mathcal{C}_1(V,V_b,E\cup\{e(v_1,v_2),e(v_2,v_1)\})}\ \textsf{c1-cons-inner}^*$$

$$\frac{\mathcal{C}_1(V,V_b,E)\quad e(v_1,v_2)}{\mathcal{C}_1(V,V_b\cup\{v_1\},E\cup\{e(v_1,v_2)\})}\ \textsf{c1-cons-boundary}^\dagger$$

* if $\{v_1,v_2\}\subseteq V, e(v_1,v_2)\notin E, e(v_2,v_1)\notin E$
and either $E=\emptyset$ or $\{v_1\}\sqsubset E$ or $\{v_2\}\sqsubset E$

† if $\{v_1,v_2\}\subseteq V, v_1\notin V_b, e(v_1,v_2)\notin E, e(v_2,v_1)\notin E$
and either $E=\emptyset$ or $\{v_1\}\sqsubset E$ or $\{v_2\}\sqsubset E$

Figure 2-3: Rules for the $\mathcal{C}_1$ complex

It is important to note that if an edge is added to the edge set as a boundary edge, the edge in the opposite direction can never be added. It is also important to note that if the set of edges E is not empty, any additional edge added to the set must share a vertex with some other edge in the set. This will be used to help guarantee connectedness.

2.3.3 The $\mathcal{C}_2$ Complex

Figure 2-4 presents the rules for construction of the $\mathcal{C}_2$ complex.

$$\frac{\mathcal{C}_1(V,V_b,E)}{\mathcal{C}_2(E,\emptyset)}\ \textsf{c2-nil}$$

$$\frac{\mathcal{C}_2(E,F)\quad f(e_1,e_2,e_3)}{\mathcal{C}_2(E,F\cup\{f(e_1,e_2,e_3)\})}\ \textsf{c2-cons}^*$$

* if $\{e_1,e_2,e_3\}\subseteq E, e_1\not\triangleleft F, e_2\not\triangleleft F, e_3\not\triangleleft F$

Figure 2-4: Rules for the $\mathcal{C}_2$ complex

Compared with the definition of the $\mathcal{C}_1$ complex, the $\mathcal{C}_2$ complex is straightforward. The only constraint we place on the c2-cons rule is that the face being added cannot share any edges with any other faces. This helps guarantee orientability.

2.4 Manifold Mesh Definition

Figure 2-5 presents the final rule for construction of a manifold mesh.

$$\frac{\mathcal{C}_0(V) \quad \mathcal{C}_1(V, V_b, E) \quad \mathcal{C}_2(E, F)}{m(V, E, F)} \text{ make-mesh}^*$$

* if $V \sqsubset E \sqsubset F$

Figure 2-5: Rule for a manifold mesh

Throughout the definition of the complexes, we have been guaranteeing that all structures in higher complexes be built out of structures in lower complexes. The mesh construction rule makes a similar constraint in the opposite direction: it requires that all of the vertexes be used in the edge set and all of the edges be used in the face set. This will be important for both connectedness and orientability.

2.4.1 Proof of Soundness

Lemma 2.1 (Soundness). *Any mesh formed by this inductive definition is a valid manifold mesh.*

Proof. Certainly this definition forms a structure of vertexes, edges, and faces. We must argue that this structure is connected, orientable, and manifold.

Connectedness is the easiest property to check. The definition of the $\mathcal{C}_1$ complex requires that any edge added after the first be connected to some other edge already in the set. Because the definition of a mesh requires that every edge be used in some face, and every vertex used in some edge, we will have a connected structure.

The structure formed will also be orientable. This is because each face is orientable since the directed edges making up each face must form a chain. Furthermore, each directed edge can only be in one face, so if two faces share an undirected edge, they will have opposite directed edges, and orientation will be preserved when crossing between these two faces.

Finally, the structure formed will be manifold. Suppose a point exists in the structure that fails to be manifold. Surely this does not occur within a face, because faces are locally homeomorphic to the plane. Furthermore this point does not occur on an edge, because every edge is part of some face. So, if the edge is a boundary edge, a neighborhood around the point will be homeomorphic to the half-plane. If the edge is an inner edge, our definitions require that a face be present on each side of the edge. So this point, too, has a local neighborhood which is homeomorphic to the plane.

Thus, the point where the structure fails to be manifold must be at a vertex v. This vertex must be a boundary vertex, for if not, the structure would certainly be manifold at the vertex. Furthermore, it must be part of two boundaries. But boundaries are cycles of directed edges. So, there are two distinct boundary edges in the structure which have a starting point of v. But this is impossible by the constraints expressed in the definition of the $\mathcal{C}_1$ complex. Namely, only one boundary edge with a particular start vertex can be added to the edge set. We reach a contradiction and conclude that the structure is manifold.

Because the structure is connected, orientable, and manifold, we conclude that the structure is a manifold mesh. □

2.4.2 Proof of Completeness

Lemma 2.2 (Completeness). *All manifold meshes are constructible by this inductive definition.*

Proof. Suppose toward contradiction that there is some mesh M which cannot be constructed from this definition. Then there must be some vertex, edge, or face in M which could not be added to the appropriate complex.

The definition of the $\mathcal{C}_0$ complex makes no restriction on what can be added (except that no vertex can be added twice), so all vertexes in M could be added by the definition.

Likewise, the $\mathcal{C}_2$ complex only stipulates that a face cannot be added if it shares an edge with another face. As we have already shown, if two faces that share an edge occur in a structure, that structure is not orientable. Since M is orientable, this must not be the case.

So then, there is some edge e in M which cannot be added to the $\mathcal{C}_1$ complex. But e cannot be an inner edge, since there are no restrictions on adding inner edges (except that edges cannot be added twice). So then e is a boundary edge. But if e cannot be added then its start vertex v_s is already in the set of boundary vertexes. So then, there exists another boundary edge which has a start vertex v_s. But since all boundaries are cycles, there are two distinct cycles in M which contain v_s. As we have already argued, this means that M is not manifold at v_s. We reach a contradiction and conclude that no such M exists. Thus, our definition is complete. □

2.4.3 A Faithful Representation

Theorem 2.3. *The inductive definition given faithfully represent manifold meshes.*

Proof. This immediately follows from Lemmas 2.1 and 2.2. □

Chapter 3

Embedding Manifold Meshes in LF

After having developed an inductive definition of manifold meshes in Chapter 2, embedding this definition in LF is quite straightforward. The building blocks of meshes (vertexes, edges, faces, vertex sets, and so on) correspond naturally to families in LF. Furthermore, the inductive definitions correspond to terms which create objects of those families.

Our only real concern is how to deal with the constraints on many of the inductive definitions. These constraints must somehow be bundled into the constructors, so that terms of a given family can only be constructed if the constraints are met.

We solve this problem by introducing two additional families, *o* and *pf*. Terms of type *o* represent propositions, which may be true or false, and terms of type *pf i* where *i* is of type *o* represent proofs of propositions. We have one proposition for each of the constraints expressed in the inductive definitions. This will be explained in greater detail in Section 3.2.

The majority of the signatures for the embedding are given in Appendix A, rather than here, because they are nearly identical to the definitions from Chapter 2.

3.1 The Mesh Families

Figure 3-1 gives the signatures for the LF families representing the building blocks of meshes.

$$
\begin{aligned}
o &:: \mathsf{type} \\
pf &:: o \Rightarrow \mathsf{type} \\
\\
num &:: \mathsf{type} \\
\\
\textit{vertex-t} &:: \mathsf{type} \\
\textit{edge-t} &:: \mathsf{type} \\
\textit{face-t} &:: \mathsf{type} \\
\\
\textit{vertex-set-t} &:: num \Rightarrow \mathsf{type} \\
\textit{edge-set-t} &:: nv{:}num \Rightarrow \textit{vertex-set-t}\ nv \Rightarrow \\
&\quad\ nbv{:}num \Rightarrow \textit{vertex-set-t}\ nbv \Rightarrow \\
&\quad\ num \Rightarrow \mathsf{type} \\
\textit{face-set-t} &:: nv{:}num \Rightarrow vs{:}\textit{vertex-set-t}\ nv \Rightarrow \\
&\quad\ nbv{:}num \Rightarrow bvs{:}\textit{vertex-set-t}\ nbv \Rightarrow \\
&\quad\ ne{:}num \Rightarrow es{:}\textit{edge-set-t}\ nv\ vs\ nbv\ bvs\ ne \Rightarrow \\
&\quad\ num \Rightarrow \mathsf{type} \\
\\
\textit{mesh-t} &:: num \Rightarrow num \Rightarrow num \Rightarrow \mathsf{type}
\end{aligned}
$$

Figure 3-1: LF families for embedding of manifold meshes

The families representing vertex sets, edge sets, and face sets are dependent on terms of other families in exactly the same way that our higher complexes were dependent on the complexes below. We also see that these families, as well has the *mesh-t* family, are dependent on terms of the *num* family. This addition allows us to express the Euler Characteristic of the mesh. We will discuss this further in Section 3.5.1

3.2 Propositions and Proof Rules

As stated previously, we want to be able to construct terms representing mesh building blocks only when the constraints from the corresponding rule in the inductive definition are met. In other words, we want to be able to construct a term only when we can prove the propositions necessary to satisfy constraints in the corresponding rule of the inductive definition.

In order to make this guarantee, we use terms of the family *pf*, a family constructed from a term of type *o*. For each particular *o* constructor (a term which has type $\cdots \Rightarrow o$), we introduce *pf o* constructors (terms which have type $\cdots \Rightarrow pfi$ where i is a term of type o) such that a term of type *pf i* can exist only if i is a true proposition. We call these terms "proof rules." So, then, we can say that the family *pf i* is inhabited only if i is provable.

We then augment our constructors for each of the mesh families with a requirement for a term of type *pf i* where i is a proposition representing a constraint that must hold. Then, as long as our proof rules are valid, we can only construct terms of mesh building blocks when the constraints are met.

The propositions expressed in the constraints of the inductive definitions are all related to set containment or equality. Because our sets are finite, proofs of these constraints are easy to build inductively.

Examples of propositions and proof rules can be seen throughout Appendix A. All terms of type $\cdots \Rightarrow o$ are propositions, and all terms of type $\cdots \Rightarrow pfi$ (where i is of type o) are proof rules.

3.3 Constructors for Mesh Objects

Using these propositions and proof rules, the constructors for mesh objects are simple translations of the inductive definitions from Chapter 2. These constructors can be seen in Appendix A.

3.4 Adequacy of Syntax

Theorem 3.1 (Adequacy). *The syntax presented here is adequate to represent manifold meshes.*

Proof. Because our definition from Chapter 2 was a faithful representation of manifold meshes, and because there is a bijection between those definitions and our constructors for terms of the families representing mesh building blocks, our syntax is adequate as long as our proof rules are correct. Our proof rules only express notions of equality,

dis-equality, set containment and set non-containment. These proof rules are defined in the conventional way for inductive proofs. Therefore, our proof rules are correct, and our syntax is adequate. □

3.5 Topology Equivalence Proofs

It is often important to argue that two meshes have the same topology. For example, in mesh optimization algorithms, output is only correct if it has the same topology as the input [6, 13, 4].

Creating families for our vertex, edge, and face sets that depend on a number indicating the cardinality of the set allows us to give proofs about topology equivalence.

3.5.1 Embedding the Euler Characteristic

As stated in Section 1.4.2, the Euler Characteristic uniquely describes the topological characteristics of any connected, compact surface. In our embedding, we consider the Euler Characteristic as a tuple of three numbers: the number of vertices, the number of (undirected) edges, and the number of faces. Then, we define an equivalence relation R on the tuples such that

$$(v_1, e_1, f_1)R(v_2, e_2, f_2) \Leftrightarrow v_1 - e_1 + f_1 = v_2 - e_2 + f_2$$

It is clear that if two tuples are in the same equivalence class, they represent the same Euler Characteristic.

Observe that the constructors for vertex, edge, and face sets increment the number corresponding to their cardinality with each additional item. (Also observe that when two opposite inner edges are added to the edge set, the count is only incremented by one, indicating the total number of undirected edges.) So then, the Euler Characteristic of a mesh is embedded in the dependencies for the family of a mesh.

From this, it is easy to build inductive proofs that two meshes have the same topology. The base case we have chosen is that meshes with tuples $(0, 0, 0)$ and $(0, 0, 0)$

have the same topology. Then, by incrementing appropriate pairs (for example, v_1 and f_2) of numbers, an inductive proof can be developed. A complete set of proof rules is given in Figure A-10 of Appendix A.

3.6 Encoding Meshes in RSP

The Edinburgh Logical Framework is a proper subset of the programming language RSP. Because of this, it is straightforward to encode our syntax in RSP. (In fact, the grammar of LF is essentially a subset of the grammar for RSP.) This encoding results in a set of dependent data types for mesh building blocks. We will discuss uses of these dependent data types in Chapter 4.

While it is trivial to take something properly embedded in LF and encode it in RSP, the usefulness of RSP in developing an embedding should not be discounted. It is tremendously useful to employ the type checker's power while developing the signatures for an embedding. We will discuss this issue further in Chapter 4 as well.

Chapter 4

Discussion and Conclusion

In this chapter, we present some of the "softer" results of our research. We begin discussing the pros and cons of program verification through type checking, as well as helpful ideas we have come across along the way. Then, we discuss potential properties of programming languages which we believe would aid program verification. We conclude with plans for future work.

4.1 Program verification through type checking

A great deal of work has been done in the field of program verification through type checking [1, 8, 9, 15]. Certainly, type checking in mainstream type-safe languages such as Java makes some specific guarantees about program correctness. The goal, however, is to push larger amounts of semantic information into the type system so that more properties can be guaranteed statically.

As this thesis has shown, dependent types are a very powerful tool for creating rich data types which make much stronger guarantees about the data. The ultimate goal, however, is to use these dependent data types in algorithms. In our case, we would like the type checker to statically verify that the algorithm produces a valid mesh, as well as statically verify properties about the topological characterization of the output. Even more ideally, we could combine type checking with coverage checking so that we could statically verify that our algorithm will both produce valid output and never fail to produce output.

4.1.1 On Verified Algorithms

It is one thing to argue that a type system can be used to statically verify that an algorithm will produce valid results. It is quite another thing to argue that a type system can be used to statically verify that an algorithm produces the *expected results*, that is, that the algorithm meets its specification.

In this thesis, we have worked quite hard toward the first goal. Yet to date, even the simplest of algorithms, called *edge split* [6], remains elusive. This algorithm takes in a mesh and an interior edge in that mesh, and creates an additional vertex in the middle of that edge. Two new edges are then created from the new vertex to the far vertexes of each of the faces bordered by the original edge.

It is quite straight forward to write this algorithm in an unverified way. However, cleaning up the proof terms so that the new structure type checks is a daunting task. Perhaps the most important observation that has come out of this work is that it appears to be exceedingly helpful to have deterministic proof rules. In other words, it is detrimental to have proof rules which allow you to prove the same proposition multiple ways. In fact, a non-deterministic proof system is nearly fatal if the programming language does not have coverage checking.

4.1.2 Discrepancy With Typical Representations

One major problem with program verification is that the data structures typically used to represent data are not well suited for program verification. For example, meshes are often stored in a *winged edge* representation [3]. This data structure leaves many of the properties of a mesh implicit, and is completely unsuitable for program verification. However, many algorithms we wish to verify are implemented using these data structures.

Representations which *are* suitable for program verification tend to duplicate large amounts of data. (This is certainly true in our system, where an ASCII representation of a mesh with two faces occupies nearly 4 gigabytes.)

These two methods of representation need to be reconciled if we ever hope to have deployable statically verified code.

4.1.3 Automated Proof Construction

We were able to implement an "algorithm" for iteratively constructing the proofs necessary to build a mesh. The algorithm takes in any proposition and attempts to return a proof of that proposition.

Using such an algorithm as a sub-procedure in other algorithms is completely contrary to the idea of verification using the type system. If such an algorithm is used, the verification given is no stronger than using *assert* calls in any mainstream language. However, development of this algorithm was crucial in finding bugs in our embedding.

When proof rules are deterministic the pattern matches required to correctly implement an automated proof construction algorithm are completely determined. If such an algorithm could be automatically produced, it would be a tremendous aid in developing embeddings.

4.2 Desirable Properties of a Language

In this section, we examine some of the properties which we feel would be useful in a programming language to be used for verification via the type system.

4.2.1 Refactoring Tools

Refactoring in a dependently typed language is very difficult due to the connection between terms and types. This is especially true when types change. However, refactoring is a crucial part of the iterative process of defining a type system for program verification. This is an area that needs to be studied further.

4.2.2 Primitive proof constructs

In developing this embedding, a tremendous amount of effort was put into expressing ideas of equality, dis-equality, and set containment. These concepts are needed in almost all verification situations (especially in algorithmic verification.) Languages

intended for program verification should attempt to streamline the process of expressing these theorems.

While we have not concerned ourselves at all with implementation efficiency, this is certainly a real world concern. As such, it would be beneficial to have verified data structures as language primitives which will operate quickly at run-time, but could be abstracted as sets for program verification.

4.2.3 Input Capabilities

Input is a tricky matter when it comes to program verification. This is because input does not enter the system with proofs about its correctness. As such, automated proof construction could be used as a means for verifying input before making it a part of the system. If the appropriate proofs could not be created, the input could be rejected, possibly with some information as to why it was rejected.

4.3 Future Work

We plan to continue to work toward a verified mesh manipulation algorithm using our system of dependent data types. Ideally, we would like to implement both *edge split* and *edge contraction* algorithms, as these two algorithms completely cover the topology-preserving modifications which can be made to meshes [6].

Additionally, we would like to work toward reducing the size of proofs within the system. This can hopefully be accomplished by factoring out repeated proofs within a term. While this will probably have no effect on the speed of type checking, it should speed up computation considerably.

Appendix A

LF Signatures for Meshes

This appendix contains all of the LF signatures for our embedding of manifold meshes. They are grouped by the structure they embed. For example, all signatures related to vertexes are grouped together.

Within each group, first the families are listed, then propositions about the families. These are followed by constructors for objects of the families, and finally by constructors for proofs of propositions.

$$
\begin{aligned}
o &:: \mathsf{type} \\
pf &:: o \Rightarrow \mathsf{type}
\end{aligned}
$$

Figure A-1: LF signatures for propositions and proofs

$$
\begin{aligned}
\mathit{num} &:: \mathsf{type} \\
\\
\mathit{num\text{-}eq} &:: \mathit{num} \Rightarrow \mathit{num} \Rightarrow o \\
\mathit{num\text{-}not\text{-}eq} &:: \mathit{num} \Rightarrow \mathit{num} \Rightarrow o \\
\\
\mathit{zero} &:: \mathit{num} \\
\mathit{succ} &:: \mathit{num} \Rightarrow \mathit{num} \\
\\
\mathit{num\text{-}eq\text{-}i} &:: \mathit{n1}{:}\mathit{num} \Rightarrow \mathit{pf}\ (\mathit{num\text{-}eq}\ \mathit{n1}\ \mathit{n1}) \\
\mathit{num\text{-}not\text{-}eq\text{-}i1} &:: \mathit{n}{:}\mathit{num} \Rightarrow \mathit{pf}\ (\mathit{num\text{-}not\text{-}eq}\ \mathit{zero}\ (\mathit{succ}\ \mathit{n})) \\
\mathit{num\text{-}not\text{-}eq\text{-}i2} &:: \mathit{n1}{:}\mathit{num} \Rightarrow \mathit{n2}{:}\mathit{num} \Rightarrow \mathit{pf}\ (\mathit{num\text{-}not\text{-}eq}\ \mathit{n1}\ \mathit{n2}) \Rightarrow \\
&\qquad \mathit{pf}\ (\mathit{num\text{-}not\text{-}eq}\ (\mathit{succ}\ \mathit{n1})\ (\mathit{succ}\ \mathit{n2})) \\
\mathit{num\text{-}not\text{-}eq\text{-}sym} &:: \mathit{n1}{:}\mathit{num} \Rightarrow \mathit{n2}{:}\mathit{num} \Rightarrow \mathit{pf}\ (\mathit{num\text{-}not\text{-}eq}\ \mathit{n1}\ \mathit{n2}) \Rightarrow \\
&\qquad \mathit{pf}\ (\mathit{num\text{-}not\text{-}eq}\ \mathit{n2}\ \mathit{n1})
\end{aligned}
$$

Figure A-2: LF signatures for numbers

$$
\begin{aligned}
\mathit{vertex\text{-}t} &:: \mathsf{type} \\
\\
\mathit{vertex\text{-}eq} &:: \mathit{vertex\text{-}t} \Rightarrow \mathit{vertex\text{-}t} \Rightarrow o \\
\mathit{vertex\text{-}not\text{-}eq} &:: \mathit{vertex\text{-}t} \Rightarrow \mathit{vertex\text{-}t} \Rightarrow o \\
\\
\mathit{vertex} &:: \mathit{num} \Rightarrow \mathit{vertex\text{-}t} \\
\\
\mathit{vertex\text{-}eq\text{-}i} &:: \mathit{v1}{:}\mathit{vertex\text{-}t} \Rightarrow \mathit{pf}\ (\mathit{vertex\text{-}eq}\ \mathit{v1}\ \mathit{v1}) \\
\mathit{vertex\text{-}not\text{-}eq\text{-}i} &:: \mathit{n1}{:}\mathit{num} \Rightarrow \mathit{n2}{:}\mathit{num} \Rightarrow \mathit{pf}\ (\mathit{num\text{-}not\text{-}eq}\ \mathit{n1}\ \mathit{n2}) \Rightarrow \\
&\qquad \mathit{pf}\ (\mathit{vertex\text{-}not\text{-}eq}\ (\mathit{vertex}\ \mathit{n1})\ (\mathit{vertex}\ \mathit{n2}))
\end{aligned}
$$

Figure A-3: LF signatures for vertexes

```
          edge-t :: type

        edge-eq :: edge-t ⇒ edge-t ⇒ o
    edge-not-eq :: edge-t ⇒ edge-t ⇒ o
     edges-join :: e1:edge-t ⇒ e2:edge-t ⇒ o
 edges-opposite :: e1:edge-t ⇒ e2:edge-t ⇒ o

           edge :: v1:vertex-t ⇒ v2:vertex-t ⇒
                   pf (vertex-not-eq v1 v2) ⇒ edge-t

      edge-eq-i :: e1:edge-t ⇒ pf (edge-eq e1 e1)
  edge-not-eq-i1 :: v1:vertex-t ⇒ v2:vertex-t ⇒
                   v3:vertex-t ⇒ v4:vertex-t ⇒
                   p1:pf (vertex-not-eq v1 v2) ⇒
                   p2:pf (vertex-not-eq v3 v4) ⇒
                   p3:pf (vertex-not-eq v1 v3) ⇒
                   pf (edge-not-eq (edge v1 v2 p1) (edge v3 v4 p2))
  edge-not-eq-i2 :: v1:vertex-t ⇒ v2:vertex-t ⇒
                   v3:vertex-t ⇒ v4:vertex-t ⇒
                   p1:pf (vertex-not-eq v1 v2) ⇒
                   p2:pf (vertex-not-eq v3 v4) ⇒
                   p3:pf (vertex-not-eq v2 v4) ⇒
                   pf (edge-not-eq (edge v1 v2 p1) (edge v3 v4 p2))
    edges-join-i :: v1:vertex-t ⇒ v2:vertex-t ⇒ v3:vertex-t ⇒
                   p1:pf (vertex-not-eq v1 v2) ⇒
                   p2:pf (vertex-not-eq v2 v3) ⇒
                   pf (edges-join (edge v1 v2 p1) (edge v2 v3 p2))
edges-opposite-i :: v1:vertex-t ⇒ v2:vertex-t ⇒
                   p1:pf (vertex-not-eq v1 v2) ⇒
                   p2:pf (vertex-not-eq v2 v1) ⇒
                   pf (edges-opposite (edge v1 v2 p1) (edge v2 v1 p2))
```

Figure A-4: LF signatures for edges

```
face-t :: type

face :: e1:edge-t ⇒ e2:edge-t ⇒ e3:edge-t ⇒
        p1:pf (edges-join e1 e2) ⇒
        p2:pf (edges-join e2 e3) ⇒
        p3:pf (edges-join e3 e1) ⇒
        face-t
```

Figure A-5: LF signatures for faces

```
vertex-set-t :: num ⇒ type

vertex-set-contains :: vertex-t ⇒ n:num ⇒ vertex-set-t n ⇒ o
vertex-set-not-contains :: vertex-t ⇒ n:num ⇒ vertex-set-t n ⇒ o

vertex-set-nil :: vertex-set-t zero
vertex-set-cons :: v:vertex-t ⇒ n:num ⇒ s:vertex-set-t n ⇒
                   p:pf (vertex-set-not-contains v n s) ⇒
                   vertex-set-t (succ n)

vertex-set-contains-i :: v:vertex-t ⇒ np:num ⇒ sp:vertex-set-t np ⇒
                         p:pf (vertex-set-not-contains v np sp) ⇒
                         pf (vertex-set-contains
                               v (succ np) (vertex-set-cons v np sp p))
vertex-set-contains-ext :: v:vertex-t ⇒ vp:vertex-t ⇒
                           np:num ⇒ sp:vertex-set-t np ⇒
                           p:pf (vertex-set-not-contains vp np sp) ⇒
                           pf (vertex-set-contains v np sp) ⇒
                           pf (vertex-set-contains
                                 v (succ np) (vertex-set-cons vp np sp p))
vertex-set-not-contains-nil :: v:vertex-t ⇒
                               pf (vertex-set-not-contains
                                     v zero vertex-set-nil)
vertex-set-not-contains-i :: v:vertex-t ⇒ vp:vertex-t ⇒
                             np:num ⇒ sp:vertex-set-t np ⇒
                             p:pf (vertex-set-not-contains vp np sp) ⇒
                             pf (vertex-set-not-contains v np sp) ⇒
                             pf (vertex-not-eq v vp) ⇒
                             pf (vertex-set-not-contains
                                   v (succ np) (vertex-set-cons vp np sp p))
```

Figure A-6: LF signatures for vertex sets

$$
\begin{aligned}
\textit{edge-set-t} :: \; & \textit{nv}{:}\textit{num} \Rightarrow \textit{vertex-set-t nv} \Rightarrow \\
& \textit{nbv}{:}\textit{num} \Rightarrow \textit{vertex-set-t nbv} \Rightarrow \\
& \textit{num} \Rightarrow \mathsf{type} \\
\\
\textit{edge-set-contains} :: \; & \textit{edge-t} \Rightarrow \textit{nv}{:}\textit{num} \Rightarrow \textit{vs}{:}\textit{vertex-set-t nv} \Rightarrow \\
& \textit{nbv}{:}\textit{num} \Rightarrow \textit{bvs}{:}\textit{vertex-set-t nbv} \Rightarrow \\
& \textit{n}{:}\textit{num} \Rightarrow \textit{edge-set-t nv vs nbv bvs n} \Rightarrow \\
& \textit{o} \\
\textit{edge-set-not-contains} :: \; & \textit{edge-t} \Rightarrow \textit{nv}{:}\textit{num} \Rightarrow \textit{vs}{:}\textit{vertex-set-t nv} \Rightarrow \\
& \textit{nbv}{:}\textit{num} \Rightarrow \textit{bvs}{:}\textit{vertex-set-t nbv} \Rightarrow \\
& \textit{n}{:}\textit{num} \Rightarrow \textit{edge-set-t nv vs nbv bvs n} \Rightarrow \\
& \textit{o} \\
\textit{edge-set-connected-to-edge} :: \; & \textit{edge-t} \Rightarrow \textit{nv}{:}\textit{num} \Rightarrow \textit{vs}{:}\textit{vertex-set-t nv} \Rightarrow \\
& \textit{nbv}{:}\textit{num} \Rightarrow \textit{bvs}{:}\textit{vertex-set-t nbv} \Rightarrow \\
& \textit{n}{:}\textit{num} \Rightarrow \textit{edge-set-t nv vs nbv bvs n} \Rightarrow \\
& \textit{o}
\end{aligned}
$$

Figure A-7: LF signatures for edge sets

```
edge-set-nil :: nv:num ⇒ vs:vertex-set-t nv ⇒
                edge-set-t nv vs zero vertex-set-nil zero
edge-set-cons-inner :: v1:vertex-t ⇒ v2:vertex-t ⇒
                p1:pf (vertex-not-eq v1 v2) ⇒
                p2:pf (vertex-not-eq v2 v1) ⇒
                nv:num ⇒ vs:vertex-set-t nv ⇒
                nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                pf (vertex-set-contains v1 nv vs) ⇒
                pf (vertex-set-contains v2 nv vs) ⇒
                n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                pf (edge-set-not-contains
                      (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                pf (edge-set-not-contains
                      (edge v2 v1 p2) nv vs nbv bvs n s) ⇒
                pf (edge-set-connected-to-edge
                      (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                edge-set-t nv vs nbv bvs (succ n)
edge-set-cons-boundary :: v1:vertex-t ⇒ v2:vertex-t ⇒
                p1:pf (vertex-not-eq v1 v2) ⇒
                p2:pf (vertex-not-eq v2 v1) ⇒ nv:num ⇒
                vs:vertex-set-t nv ⇒ nbv:num ⇒
                bvs:vertex-set-t nbv ⇒
                pf (vertex-set-contains v1 nv vs) ⇒
                pf (vertex-set-contains v2 nv vs) ⇒
                n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                pf (edge-set-not-contains
                      (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                pf (edge-set-not-contains
                      (edge v2 v1 p2) nv vs nbv bvs n s) ⇒
                pf (edge-set-connected-to-edge
                      (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                p3:pf (vertex-set-not-contains v1 nbv bvs) ⇒
                edge-set-t nv vs (succ nbv)
                      (vertex-set-cons v1 nbv bvs p3) (succ n)
```

LF signatures for edge sets (continued)

```
edge-set-contains-i-inner1 :: v1:vertex-t ⇒ v2:vertex-t ⇒
                              p1:pf (vertex-not-eq v1 v2) ⇒
                              p2:pf (vertex-not-eq v2 v1) ⇒
                              nv:num ⇒ vs:vertex-set-t nv ⇒
                              nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                              p3:pf (vertex-set-contains v1 nv vs) ⇒
                              p4:pf (vertex-set-contains v2 nv vs) ⇒
                              n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                              p5:pf (edge-set-not-contains
                                   (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                              p6:pf (edge-set-not-contains
                                   (edge v2 v1 p2) nv vs nbv bvs n s) ⇒
                              p7:pf (edge-set-connected-to-edge
                                   (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                              pf (edge-set-contains
                                   (edge v1 v2 p1) nv vs nbv bvs (succ n)
                                   (edge-set-cons-inner
                                        v1 v2 p1 p2 nv vs nbv bvs
                                        p3 p4 n s p5 p6 p7))
edge-set-contains-i-inner2 :: v1:vertex-t ⇒ v2:vertex-t ⇒
                              p1:pf (vertex-not-eq v1 v2) ⇒
                              p2:pf (vertex-not-eq v2 v1) ⇒
                              nv:num ⇒ vs:vertex-set-t nv ⇒
                              nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                              p3:pf (vertex-set-contains v1 nv vs) ⇒
                              p4:pf (vertex-set-contains v2 nv vs) ⇒
                              n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                              p5:pf (edge-set-not-contains
                                   (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                              p6:pf (edge-set-not-contains
                                   (edge v2 v1 p2) nv vs nbv bvs n s) ⇒
                              p7:pf (edge-set-connected-to-edge
                                   (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                              pf (edge-set-contains
                                   (edge v2 v1 p2) nv vs nbv bvs (succ n)
                                   (edge-set-cons-inner
                                        v1 v2 p1 p2 nv vs nbv bvs
                                        p3 p4 n s p5 p6 p7))
```

LF signatures for edge sets (continued)

```
edge-set-contains-i-boundary :: v1:vertex-t ⇒ v2:vertex-t ⇒
                                p1:pf (vertex-not-eq v1 v2) ⇒
                                p2:pf (vertex-not-eq v2 v1) ⇒
                                nv:num ⇒ vs:vertex-set-t nv ⇒
                                nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                                p3:pf (vertex-set-contains v1 nv vs) ⇒
                                p4:pf (vertex-set-contains v2 nv vs) ⇒
                                n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                                p5:pf (edge-set-not-contains
                                    (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                                p6:pf (edge-set-not-contains
                                    (edge v2 v1 p2) nv vs nbv bvs n s) ⇒
                                p7:pf (edge-set-connected-to-edge
                                    (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                                p8:pf (vertex-set-not-contains v1 nbv bvs) ⇒
                                pf (edge-set-contains
                                    (edge v1 v2 p1) nv vs
                                    (succ nbv)
                                    (vertex-set-cons v1 nbv bvs p8)
                                    (succ n)
                                    (edge-set-cons-boundary
                                        v1 v2 p1 p2 nv vs nbv bvs
                                        p3 p4 n s p5 p6 p7 p8))
```

LF signatures for edge sets (continued)

```
edge-set-contains-ext-inner :: e:edge-t ⇒ v1:vertex-t ⇒ v2:vertex-t ⇒
                               p1:pf (vertex-not-eq v1 v2) ⇒
                               p2:pf (vertex-not-eq v2 v1) ⇒
                               nv:num ⇒ vs:vertex-set-t nv ⇒
                               nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                               p3:pf (vertex-set-contains v1 nv vs) ⇒
                               p4:pf (vertex-set-contains v2 nv vs) ⇒
                               n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                               p5:pf (edge-set-not-contains
                                    (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                               p6:pf (edge-set-not-contains
                                    (edge v2 v1 p2) nv vs nbv bvs n s) ⇒
                               p7:pf (edge-set-connected-to-edge
                                    (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                               pf (edge-set-contains e nv vs nbv bvs n s) ⇒
                               pf (edge-set-contains e nv vs nbv bvs (succ n)
                                    (edge-set-cons-inner
                                         v1 v2 p1 p2 nv vs nbv bvs
                                         p3 p4 n s p5 p6 p7))
```

LF signatures for edge sets (continued)

```
edge-set-contains-ext-boundary :: e:edge-t ⇒ v1:vertex-t ⇒ v2:vertex-t ⇒
                                  p1:pf (vertex-not-eq v1 v2) ⇒
                                  p2:pf (vertex-not-eq v2 v1) ⇒
                                  nv:num ⇒ vs:vertex-set-t nv ⇒
                                  nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                                  p3:pf (vertex-set-contains v1 nv vs) ⇒
                                  p4:pf (vertex-set-contains v2 nv vs) ⇒
                                  n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                                  p5:pf (edge-set-not-contains
                                      (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                                  p6:pf (edge-set-not-contains
                                      (edge v2 v1 p2) nv vs nbv bvs n s) ⇒
                                  p7:pf (edge-set-connected-to-edge
                                      (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                                  p8:pf (vertex-set-not-contains v1 nbv bvs) ⇒
                                  pf (edge-set-contains e nv vs nbv bvs n s) ⇒
                                  pf (edge-set-contains e nv vs
                                      (succ nbv) (vertex-set-cons v1 nbv bvs p8)
                                      (succ n)
                                      (edge-set-cons-boundary
                                          v1 v2 p1 p2 nv vs nbv bvs
                                          p3 p4 n s p5 p6 p7 p8))
```

LF signatures for edge sets (continued)

```
edge-set-not-contains-nil :: e:edge-t ⇒ nv:num ⇒ vs:vertex-set-t nv ⇒
                             pf (edge-set-not-contains
                                  e nv vs zero vertex-set-nil zero
                                  (edge-set-nil nv vs))
edge-set-not-contains-i-inner :: e:edge-t ⇒ v1:vertex-t ⇒ v2:vertex-t ⇒
                             p1:pf (vertex-not-eq v1 v2) ⇒
                             p2:pf (vertex-not-eq v2 v1) ⇒
                             nv:num ⇒ vs:vertex-set-t nv ⇒
                             nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                             p3:pf (vertex-set-contains v1 nv vs) ⇒
                             p4:pf (vertex-set-contains v2 nv vs) ⇒
                             n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                             p5:pf (edge-set-not-contains
                                  (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                             p6:pf (edge-set-not-contains
                                  (edge v2 v1 p2) nv vs nbv bvs n s) ⇒
                             p7:pf (edge-set-connected-to-edge
                                  (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                             pf (edge-set-not-contains
                                  e nv vs nbv bvs n s) ⇒
                             pf (edge-not-eq e (edge v1 v2 p1)) ⇒
                             pf (edge-not-eq e (edge v2 v1 p2)) ⇒
                             pf (edge-set-not-contains e nv vs nbv bvs
                                  (succ n)
                                  (edge-set-cons-inner
                                       v1 v2 p1 p2 nv vs nbv bvs
                                       p3 p4 n s p5 p6 p7))
```

LF signatures for edge sets (continued)

```
edge-set-not-contains-i-boundary :: e:edge-t ⇒ v1:vertex-t ⇒ v2:vertex-t ⇒
                                    p1:pf (vertex-not-eq v1 v2) ⇒
                                    p2:pf (vertex-not-eq v2 v1) ⇒
                                    nv:num ⇒ vs:vertex-set-t nv ⇒
                                    nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                                    p3:pf (vertex-set-contains v1 nv vs) ⇒
                                    p4:pf (vertex-set-contains v2 nv vs) ⇒
                                    n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                                    p5:pf (edge-set-not-contains
                                        (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                                    p6:pf (edge-set-not-contains
                                        (edge v2 v1 p2) nv vs nbv bvs n s) ⇒
                                    p7:pf (edge-set-connected-to-edge
                                        (edge v1 v2 p1) nv vs nbv bvs n s) ⇒
                                    p8:pf (vertex-set-not-contains
                                        v1 nbv bvs) ⇒
                                    pf (edge-set-not-contains
                                        e nv vs nbv bvs n s) ⇒
                                    pf (edge-not-eq e (edge v1 v2 p1)) ⇒
                                    pf (edge-set-not-contains e nv vs
                                        (succ nbv)
                                        (vertex-set-cons v1 nbv bvs p8)
                                        (succ n)
                                        (edge-set-cons-boundary
                                            v1 v2 p1 p2 nv vs nbv bvs
                                            p3 p4 n s p5 p6 p7 p8))
```

LF signatures for edge sets (continued)

```
edge-set-connected-to-edge-nil :: e:edge-t ⇒
                                  nv:num ⇒ vs:vertex-set-t nv ⇒
                                  pf (edge-set-connected-to-edge
                                        e nv vs zero vertex-set-nil zero
                                        (edge-set-nil nv vs))
edge-set-connected-to-edge-i1 :: v1:vertex-t ⇒ v2:vertex-t ⇒ v3:vertex-t ⇒
                                  p1:pf (vertex-not-eq v1 v2) ⇒
                                  p2:pf (vertex-not-eq v1 v3) ⇒
                                  nv:num ⇒ vs:vertex-set-t nv ⇒
                                  nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                                  n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                                  pf (edge-set-contains
                                        (edge v1 v3 p2) nv vs nbv bvs n s) ⇒
                                  pf (edge-set-connected-to-edge
                                        (edge v1 v2 p1) nv vs nbv bvs n s)
edge-set-connected-to-edge-i2 :: v1:vertex-t ⇒ v2:vertex-t ⇒ v3:vertex-t ⇒
                                  p1:pf (vertex-not-eq v1 v2) ⇒
                                  p2:pf (vertex-not-eq v3 v1) ⇒
                                  nv:num ⇒ vs:vertex-set-t nv ⇒
                                  nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                                  n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                                  pf (edge-set-contains
                                        (edge v3 v1 p2) nv vs nbv bvs n s) ⇒
                                  pf (edge-set-connected-to-edge
                                        (edge v1 v2 p1) nv vs nbv bvs n s)
edge-set-connected-to-edge-sym :: v1:vertex-t ⇒ v2:vertex-t ⇒
                                  p1:pf (vertex-not-eq v1 v2) ⇒
                                  p2:pf (vertex-not-eq v2 v1) ⇒
                                  nv:num ⇒ vs:vertex-set-t nv ⇒
                                  nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                                  n:num ⇒ s:edge-set-t nv vs nbv bvs n ⇒
                                  pf (edge-set-connected-to-edge
                                        (edge v2 v1 p2) nv vs nbv bvs n s) ⇒
                                  pf (edge-set-connected-to-edge
                                        (edge v1 v2 p1) nv vs nbv bvs n s)
```

LF signatures for edge sets (continued)

$$
\begin{aligned}
\textit{face-set-t} :: \; & \textit{nv}{:}\textit{num} \Rightarrow \textit{vs}{:}\textit{vertex-set-t nv} \Rightarrow \\
& \textit{nbv}{:}\textit{num} \Rightarrow \textit{bvs}{:}\textit{vertex-set-t nbv} \Rightarrow \\
& \textit{ne}{:}\textit{num} \Rightarrow \textit{es}{:}\textit{edge-set-t nv vs nbv bvs ne} \Rightarrow \\
& \textit{num} \Rightarrow \mathsf{type} \\
\\
\textit{face-set-cfwe} :: \; & \textit{edge-t} \Rightarrow \textit{nv}{:}\textit{num} \Rightarrow \textit{vs}{:}\textit{vertex-set-t nv} \Rightarrow \\
& \textit{nbv}{:}\textit{num} \Rightarrow \textit{bvs}{:}\textit{vertex-set-t nbv} \Rightarrow \\
& \textit{ne}{:}\textit{num} \Rightarrow \textit{es}{:}\textit{edge-set-t nv vs nbv bvs ne} \Rightarrow \\
& \textit{n}{:}\textit{num} \Rightarrow \textit{face-set-t nv vs nbv bvs ne es n} \Rightarrow \\
& \textit{o} \\
\textit{face-set-not-cfwe} :: \; & \textit{edge-t} \Rightarrow \textit{nv}{:}\textit{num} \Rightarrow \textit{vs}{:}\textit{vertex-set-t nv} \Rightarrow \\
& \textit{nbv}{:}\textit{num} \Rightarrow \textit{bvs}{:}\textit{vertex-set-t nbv} \Rightarrow \\
& \textit{ne}{:}\textit{num} \Rightarrow \textit{es}{:}\textit{edge-set-t nv vs nbv bvs ne} \Rightarrow \\
& \textit{n}{:}\textit{num} \Rightarrow \textit{face-set-t nv vs nbv bvs ne es n} \Rightarrow \\
& \textit{o}
\end{aligned}
$$

Figure A-8: LF signatures for face sets

```
face-set-nil :: nv:num ⇒ vs:vertex-set-t nv ⇒
                nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                ne:num ⇒ es:edge-set-t nv vs nbv bvs ne ⇒
                face-set-t nv vs nbv bvs ne es zero
face-set-cons :: e1:edge-t ⇒ e2:edge-t ⇒ e3:edge-t ⇒
                p1:pf (edges-join e1 e2) ⇒
                p2:pf (edges-join e2 e3) ⇒
                p3:pf (edges-join e3 e1) ⇒
                nv:num ⇒ vs:vertex-set-t nv ⇒
                nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                ne:num ⇒ es:edge-set-t nv vs nbv bvs ne ⇒
                n:num ⇒ s:face-set-t nv vs nbv bvs ne es n ⇒
                p4:pf (edge-set-contains
                      e1 nv vs nbv bvs ne es) ⇒
                p5:pf (edge-set-contains
                      e2 nv vs nbv bvs ne es) ⇒
                p6:pf (edge-set-contains
                      e3 nv vs nbv bvs ne es) ⇒
                p7:pf (face-set-not-cfwe
                      e1 nv vs nbv bvs ne es n s) ⇒
                p8:pf (face-set-not-cfwe
                      e2 nv vs nbv bvs ne es n s) ⇒
                p9:pf (face-set-not-cfwe
                      e3 nv vs nbv bvs ne es n s) ⇒
                face-set-t nv vs nbv bvs ne es (succ n)
```

LF signatures for face sets (continued)

```
face-set-cfwe-i1 :: e1:edge-t ⇒ e2:edge-t ⇒ e3:edge-t ⇒
                    p1:pf (edges-join e1 e2) ⇒
                    p2:pf (edges-join e2 e3) ⇒
                    p3:pf (edges-join e3 e1) ⇒
                    nv:num ⇒ vs:vertex-set-t nv ⇒
                    nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                    ne:num ⇒ es:edge-set-t nv vs nbv bvs ne ⇒
                    n:num ⇒ s:face-set-t nv vs nbv bvs ne es n ⇒
                    p4:pf (edge-set-contains
                          e1 nv vs nbv bvs ne es) ⇒
                    p5:pf (edge-set-contains
                          e2 nv vs nbv bvs ne es) ⇒
                    p6:pf (edge-set-contains
                          e3 nv vs nbv bvs ne es) ⇒
                    p7:pf (face-set-not-cfwe
                          e1 nv vs nbv bvs ne es n s) ⇒
                    p8:pf (face-set-not-cfwe
                          e2 nv vs nbv bvs ne es n s) ⇒
                    p9:pf (face-set-not-cfwe
                          e3 nv vs nbv bvs ne es n s) ⇒
                    pf (face-set-cfwe
                          e1 nv vs nbv bvs ne es
                          (succ n)
                          (face-set-cons
                                e1 e2 e3 p1 p2 p3
                                nv vs nbv bvs ne es n s
                                p4 p5 p6 p7 p8 p9))
```

LF signatures for face sets (continued)

```
face-set-cfwe-i2 :: e1:edge-t ⇒ e2:edge-t ⇒ e3:edge-t ⇒
                    p1:pf (edges-join e1 e2) ⇒
                    p2:pf (edges-join e2 e3) ⇒
                    p3:pf (edges-join e3 e1) ⇒
                    nv:num ⇒ vs:vertex-set-t nv ⇒
                    nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                    ne:num ⇒ es:edge-set-t nv vs nbv bvs ne ⇒
                    n:num ⇒ s:face-set-t nv vs nbv bvs ne es n ⇒
                    p4:pf (edge-set-contains
                          e1 nv vs nbv bvs ne es) ⇒
                    p5:pf (edge-set-contains
                          e2 nv vs nbv bvs ne es) ⇒
                    p6:pf (edge-set-contains
                          e3 nv vs nbv bvs ne es) ⇒
                    p7:pf (face-set-not-cfwe
                          e1 nv vs nbv bvs ne es n s) ⇒
                    p8:pf (face-set-not-cfwe
                          e2 nv vs nbv bvs ne es n s) ⇒
                    p9:pf (face-set-not-cfwe
                          e3 nv vs nbv bvs ne es n s) ⇒
                    pf (face-set-cfwe
                          e2 nv vs nbv bvs ne es
                          (succ n)
                          (face-set-cons
                                e1 e2 e3 p1 p2 p3
                                nv vs nbv bvs ne es n s
                                p4 p5 p6 p7 p8 p9))
```

LF signatures for face sets (continued)

```
face-set-cfwe-i3 :: e1:edge-t ⇒ e2:edge-t ⇒ e3:edge-t ⇒
                    p1:pf (edges-join e1 e2) ⇒
                    p2:pf (edges-join e2 e3) ⇒
                    p3:pf (edges-join e3 e1) ⇒
                    nv:num ⇒ vs:vertex-set-t nv ⇒
                    nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                    ne:num ⇒ es:edge-set-t nv vs nbv bvs ne ⇒
                    n:num ⇒ s:face-set-t nv vs nbv bvs ne es n ⇒
                    p4:pf (edge-set-contains
                          e1 nv vs nbv bvs ne es) ⇒
                    p5:pf (edge-set-contains
                          e2 nv vs nbv bvs ne es) ⇒
                    p6:pf (edge-set-contains
                          e3 nv vs nbv bvs ne es) ⇒
                    p7:pf (face-set-not-cfwe
                          e1 nv vs nbv bvs ne es n s) ⇒
                    p8:pf (face-set-not-cfwe
                          e2 nv vs nbv bvs ne es n s) ⇒
                    p9:pf (face-set-not-cfwe
                          e3 nv vs nbv bvs ne es n s) ⇒
                    pf (face-set-cfwe
                          e3 nv vs nbv bvs ne es
                          (succ n)
                          (face-set-cons
                                e1 e2 e3 p1 p2 p3
                                nv vs nbv bvs ne es n s
                                p4 p5 p6 p7 p8 p9))
```

LF signatures for face sets (continued)

```
face-set-cfwe-ext :: e:edge-t ⇒ e1:edge-t ⇒ e2:edge-t ⇒ e3:edge-t ⇒
                     p1:pf (edges-join e1 e2) ⇒
                     p2:pf (edges-join e2 e3) ⇒
                     p3:pf (edges-join e3 e1) ⇒
                     nv:num ⇒ vs:vertex-set-t nv ⇒
                     nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                     ne:num ⇒ es:edge-set-t nv vs nbv bvs ne ⇒
                     n:num ⇒ s:face-set-t nv vs nbv bvs ne es n ⇒
                     p4:pf (edge-set-contains
                          e1 nv vs nbv bvs ne es) ⇒
                     p5:pf (edge-set-contains
                          e2 nv vs nbv bvs ne es) ⇒
                     p6:pf (edge-set-contains
                          e3 nv vs nbv bvs ne es) ⇒
                     p7:pf (face-set-not-cfwe
                          e1 nv vs nbv bvs ne es n s) ⇒
                     p8:pf (face-set-not-cfwe
                          e2 nv vs nbv bvs ne es n s) ⇒
                     p9:pf (face-set-not-cfwe
                          e3 nv vs nbv bvs ne es n s) ⇒
                     pf (face-set-cfwe
                          e nv vs nbv bvs ne es n s) ⇒
                     pf (face-set-cfwe
                          e nv vs nbv bvs ne es (succ n)
                          (face-set-cons
                               e1 e2 e3 p1 p2 p3
                               nv vs nbv bvs ne es n s
                               p4 p5 p6 p7 p8 p9))
```

LF signatures for face sets (continued)

```
face-set-not-cfwe-nil :: e:edge-t ⇒
                         nv:num ⇒ vs:vertex-set-t nv ⇒
                         nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                         ne:num ⇒ es:edge-set-t nv vs nbv bvs ne ⇒
                         pf (face-set-not-cfwe
                               e nv vs nbv bvs ne es zero
                               (face-set-nil nv vs nbv bvs ne es))
  face-set-not-cfwe-i :: e:edge-t ⇒ e1:edge-t ⇒ e2:edge-t ⇒ e3:edge-t ⇒
                         p1:pf (edges-join e1 e2) ⇒
                         p2:pf (edges-join e2 e3) ⇒
                         p3:pf (edges-join e3 e1) ⇒
                         nv:num ⇒ vs:vertex-set-t nv ⇒
                         nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                         ne:num ⇒ es:edge-set-t nv vs nbv bvs ne ⇒
                         n:num ⇒ s:face-set-t nv vs nbv bvs ne es n ⇒
                         p4:pf (edge-set-contains
                               e1 nv vs nbv bvs ne es) ⇒
                         p5:pf (edge-set-contains
                               e2 nv vs nbv bvs ne es) ⇒
                         p6:pf (edge-set-contains
                               e3 nv vs nbv bvs ne es) ⇒
                         p7:pf (face-set-not-cfwe
                               e1 nv vs nbv bvs ne es n s) ⇒
                         p8:pf (face-set-not-cfwe
                               e2 nv vs nbv bvs ne es n s) ⇒
                         p9:pf (face-set-not-cfwe
                               e3 nv vs nbv bvs ne es n s) ⇒
                         pf (edge-not-eq e e1) ⇒
                         pf (edge-not-eq e e2) ⇒
                         pf (edge-not-eq e e3) ⇒
                         pf (face-set-not-cfwe
                               e nv vs nbv bvs ne es n s) ⇒
                         pf (face-set-not-cfwe
                               e nv vs nbv bvs ne es (succ n)
                               (face-set-cons
                                     e1 e2 e3 p1 p2 p3
                                     nv vs nbv bvs ne es n s
                                     p4 p5 p6 p7 p8 p9))
```

LF signatures for face sets (continued)

```
          mesh-t :: num ⇒ num ⇒ num ⇒ type

all-vertexes-in-es :: nv-v:num ⇒ vs-v:vertex-set-t nv-v ⇒
                      nv-e:num ⇒ vs-e:vertex-set-t nv-e ⇒
                      nbv-e:num ⇒ bvs-e:vertex-set-t nbv-e ⇒
                      ne-e:num ⇒ es-e:edge-set-t nv-e vs-e nbv-e bvs-e ne-e ⇒
                      o
   all-edges-in-fs :: nv-e:num ⇒ vs-e:vertex-set-t nv-e ⇒
                      nbv-e:num ⇒ bvs-e:vertex-set-t nbv-e ⇒
                      ne-e:num ⇒ es-e:edge-set-t nv-e vs-e nbv-e bvs-e ne-e ⇒
                      nv-f:num ⇒ vs-f:vertex-set-t nv-f ⇒
                      nbv-f:num ⇒ bvs-f:vertex-set-t nbv-f ⇒
                      ne-f:num ⇒ es-f:edge-set-t nv-f vs-f nbv-f bvs-f ne-f ⇒
                      nf-f:num ⇒
                      fs-f:face-set-t nv-f vs-f nbv-f bvs-f ne-f es-f nf-f ⇒
                      o

          mesh :: nv:num ⇒ vs:vertex-set-t nv ⇒
                  nbv:num ⇒ bvs:vertex-set-t nbv ⇒
                  ne:num ⇒ es:edge-set-t nv vs nbv bvs ne ⇒
                  nf:num ⇒ fs:face-set-t nv vs nbv bvs ne es nf ⇒
                  pf (all-vertexes-in-es
                        nv vs
                        nv vs nbv bvs ne es) ⇒
                  pf (all-edges-in-fs
                        nv vs nbv bvs ne es
                        nv vs nbv bvs ne es nf fs) ⇒
                  mesh-t nv ne nf
```

Figure A-9: LF signatures for meshes

```
all-vertexes-in-es-nil :: nv-e:num ⇒ vs-e:vertex-set-t nv-e ⇒
                          nbv-e:num ⇒ bvs-e:vertex-set-t nbv-e ⇒
                          ne-e:num ⇒
                          es-e:edge-set-t nv-e vs-e nbv-e bvs-e ne-e ⇒
                          pf (all-vertexes-in-es
                               zero vertex-set-nil
                               nv-e vs-e nbv-e bvs-e ne-e es-e)
all-vertexes-in-es-ext-l :: v1:vertex-t ⇒ v2:vertex-t ⇒
                            p1:pf (vertex-not-eq v1 v2) ⇒
                            nv-v:num ⇒ vs-v:vertex-set-t nv-v ⇒
                            p2:pf (vertex-set-not-contains
                                 v1 nv-v vs-v) ⇒
                            nv-e:num ⇒ vs-e:vertex-set-t nv-e ⇒
                            nbv-e:num ⇒ bvs-e:vertex-set-t nbv-e ⇒
                            ne-e:num ⇒
                            es-e:edge-set-t nv-e vs-e nbv-e bvs-e ne-e ⇒
                            pf (edge-set-contains
                                 (edge v1 v2 p1)
                                 nv-e vs-e nbv-e bvs-e ne-e es-e) ⇒
                            pf (all-vertexes-in-es
                                 nv-v vs-v nv-e vs-e nbv-e bvs-e ne-e es-e) ⇒
                            pf (all-vertexes-in-es
                                 (succ nv-v)
                                 (vertex-set-cons v1 nv-v vs-v p2)
                                 nv-e vs-e nbv-e bvs-e ne-e es-e)
```

LF signatures for meshes (continued)

```
all-vertexes-in-es-ext-r :: v1:vertex-t ⇒ v2:vertex-t ⇒
                            p1:pf (vertex-not-eq v1 v2) ⇒
                            nv-v:num ⇒ vs-v:vertex-set-t nv-v ⇒
                            p2:pf (vertex-set-not-contains
                                  v2 nv-v vs-v) ⇒
                            nv-e:num ⇒ vs-e:vertex-set-t nv-e ⇒
                            nbv-e:num ⇒ bvs-e:vertex-set-t nbv-e ⇒
                            ne-e:num ⇒
                            es-e:edge-set-t nv-e vs-e nbv-e bvs-e ne-e ⇒
                            pf (edge-set-contains
                                  (edge v1 v2 p1)
                                  nv-e vs-e nbv-e bvs-e ne-e es-e) ⇒
                            pf (all-vertexes-in-es
                                  nv-v vs-v nv-e vs-e nbv-e bvs-e ne-e es-e) ⇒
                            pf (all-vertexes-in-es
                                  (succ nv-v)
                                  (vertex-set-cons v2 nv-v vs-v p2)
                                  nv-e vs-e nbv-e bvs-e ne-e es-e)
     all-edges-in-fs-nil :: nv-e:num ⇒ vs-e:vertex-set-t nv-e ⇒
                            nv-f:num ⇒ vs-f:vertex-set-t nv-f ⇒
                            nbv-f:num ⇒ bvs-f:vertex-set-t nbv-f ⇒
                            ne-f:num ⇒
                            es-f:edge-set-t nv-f vs-f nbv-f bvs-f ne-f ⇒
                            nf-f:num ⇒
                            fs-f:face-set-t nv-f vs-f nbv-f bvs-f ne-f es-f nf-f ⇒
                            pf (all-edges-in-fs
                                  nv-e vs-e zero vertex-set-nil
                                  zero (edge-set-nil nv-e vs-e)
                                  nv-f vs-f nbv-f bvs-f ne-f es-f nf-f fs-f)
```

LF signatures for meshes (continued)

```
all-edges-in-fs-ext-in :: v1:vertex-t ⇒ v2:vertex-t ⇒
                          p1:pf (vertex-not-eq v1 v2) ⇒
                          p2:pf (vertex-not-eq v2 v1) ⇒
                          nv-e:num ⇒ vs-e:vertex-set-t nv-e ⇒
                          nbv-e:num ⇒ bvs-e:vertex-set-t nbv-e ⇒
                          p3:pf (vertex-set-contains v1 nv-e vs-e) ⇒
                          p4:pf (vertex-set-contains v2 nv-e vs-e) ⇒
                          ne-e:num ⇒
                          es-e:edge-set-t nv-e vs-e nbv-e bvs-e ne-e ⇒
                          p5:pf (edge-set-not-contains
                                (edge v1 v2 p1) nv-e vs-e nbv-e bvs-e ne-e es-e) ⇒
                          p6:pf (edge-set-not-contains
                                (edge v2 v1 p2) nv-e vs-e nbv-e bvs-e ne-e es-e) ⇒
                          p7:pf (edge-set-connected-to-edge
                                (edge v1 v2 p1) nv-e vs-e nbv-e bvs-e ne-e es-e) ⇒
                          nv-f:num ⇒ vs-f:vertex-set-t nv-f ⇒
                          nbv-f:num ⇒ bvs-f:vertex-set-t nbv-f ⇒
                          ne-f:num ⇒
                          es-f:edge-set-t nv-f vs-f nbv-f bvs-f ne-f ⇒
                          nf-f:num ⇒
                          fs-f:face-set-t nv-f vs-f nbv-f bvs-f ne-f es-f nf-f ⇒
                          pf (face-set-cfwe
                                (edge v1 v2 p1)
                                nv-f vs-f nbv-f bvs-f ne-f es-f nf-f fs-f) ⇒
                          pf (face-set-cfwe
                                (edge v2 v1 p2)
                                nv-f vs-f nbv-f bvs-f ne-f es-f nf-f fs-f) ⇒
                          pf (all-edges-in-fs
                                nv-e vs-e nbv-e bvs-e ne-e es-e
                                nv-f vs-f nbv-f bvs-f ne-f es-f nf-f fs-f) ⇒
                          pf (all-edges-in-fs
                                nv-e vs-e nbv-e bvs-e
                                (succ ne-e)
                                (edge-set-cons-inner
                                      v1 v2 p1 p2
                                      nv-e vs-e nbv-e bvs-e p3 p4
                                      ne-e es-e p5 p6 p7)
                                nv-f vs-f nbv-f bvs-f ne-f es-f nf-f fs-f)
```

LF signatures for meshes (continued)

```
all-edges-in-fs-ext-bdry :: v1:vertex-t ⇒ v2:vertex-t ⇒
                            p1:pf (vertex-not-eq v1 v2) ⇒
                            p2:pf (vertex-not-eq v2 v1) ⇒
                            nv-e:num ⇒ vs-e:vertex-set-t nv-e ⇒
                            nbv-e:num ⇒ bvs-e:vertex-set-t nbv-e ⇒
                            p3:pf (vertex-set-contains v1 nv-e vs-e) ⇒
                            p4:pf (vertex-set-contains v2 nv-e vs-e) ⇒
                            ne-e:num ⇒
                            es-e:edge-set-t nv-e vs-e nbv-e bvs-e ne-e ⇒
                            p5:pf (edge-set-not-contains
                                (edge v1 v2 p1) nv-e vs-e nbv-e bvs-e ne-e es-e) ⇒
                            p6:pf (edge-set-not-contains
                                (edge v2 v1 p2) nv-e vs-e nbv-e bvs-e ne-e es-e) ⇒
                            p7:pf (edge-set-connected-to-edge
                                (edge v1 v2 p1) nv-e vs-e nbv-e bvs-e ne-e es-e) ⇒
                            p8:pf (vertex-set-not-contains v1 nbv-e bvs-e) ⇒
                            nv-f:num ⇒ vs-f:vertex-set-t nv-f ⇒
                            nbv-f:num ⇒ bvs-f:vertex-set-t nbv-f ⇒
                            ne-f:num ⇒
                            es-f:edge-set-t nv-f vs-f nbv-f bvs-f ne-f ⇒
                            nf-f:num ⇒
                            fs-f:face-set-t nv-f vs-f nbv-f bvs-f ne-f es-f nf-f ⇒
                            pf (face-set-cfwe
                                (edge v1 v2 p1)
                                nv-f vs-f nbv-f bvs-f ne-f es-f nf-f fs-f) ⇒
                            pf (all-edges-in-fs
                                nv-e vs-e nbv-e bvs-e ne-e es-e
                                nv-f vs-f nbv-f bvs-f ne-f es-f nf-f fs-f) ⇒
                            pf (all-edges-in-fs
                                nv-e vs-e
                                (succ nbv-e) (vertex-set-cons v1 nbv-e bvs-e p8)
                                (succ ne-e)
                                (edge-set-cons-boundary
                                    v1 v2 p1 p2 nv-e vs-e nbv-e bvs-e p3 p4
                                    ne-e es-e p5 p6 p7 p8)
                                nv-f vs-f nbv-f bvs-f ne-f es-f nf-f fs-f)
```

LF signatures for meshes (continued)

```
same-topology :: nv1:num ⇒ ne1:num ⇒ nf1:num ⇒
                 nv2:num ⇒ ne2:num ⇒ nf2:num ⇒ o

same-topology-zero :: pf (same-topology
                          zero zero zero
                          zero zero zero)
same-topology-sym :: nv1:num ⇒ ne1:num ⇒ nf1:num ⇒
                     nv2:num ⇒ ne2:num ⇒ nf2:num ⇒
                     pf (same-topology
                         nv1 ne1 nf1
                         nv2 ne2 nf2) ⇒
                     pf (same-topology
                         nv2 ne2 nf2
                         nv1 ne1 nf1)
same-topology-v1-v2 :: nv1:num ⇒ ne1:num ⇒ nf1:num ⇒
                       nv2:num ⇒ ne2:num ⇒ nf2:num ⇒
                       pf (same-topology
                           nv1 ne1 nf1
                           nv2 ne2 nf2) ⇒
                       pf (same-topology
                           (succ nv1) ne1 nf1
                           (succ nv2) ne2 nf2)
same-topology-e1-e2 :: nv1:num ⇒ ne1:num ⇒ nf1:num ⇒
                       nv2:num ⇒ ne2:num ⇒ nf2:num ⇒
                       pf (same-topology
                           nv1 ne1 nf1
                           nv2 ne2 nf2) ⇒
                       pf (same-topology
                           nv1 (succ ne1) nf1
                           nv2 (succ ne2) nf2)
same-topology-f1-f2 :: nv1:num ⇒ ne1:num ⇒ nf1:num ⇒
                       nv2:num ⇒ ne2:num ⇒ nf2:num ⇒
                       pf (same-topology
                           nv1 ne1 nf1
                           nv2 ne2 nf2) ⇒
                       pf (same-topology
                           nv1 ne1 (succ nf1)
                           nv2 ne2 (succ nf2))
```

Figure A-10: LF signatures for topology equivalence judgements

$$
\begin{array}{l}
\textit{same-topology-v1-e1} :: \textit{nv1}{:}\textit{num} \Rightarrow \textit{ne1}{:}\textit{num} \Rightarrow \textit{nf1}{:}\textit{num} \Rightarrow \\
\qquad \textit{nv2}{:}\textit{num} \Rightarrow \textit{ne2}{:}\textit{num} \Rightarrow \textit{nf2}{:}\textit{num} \Rightarrow \\
\qquad \textit{pf}\ (\textit{same-topology} \\
\qquad\qquad \textit{nv1}\ \textit{ne1}\ \textit{nf1} \\
\qquad\qquad \textit{nv2}\ \textit{ne2}\ \textit{nf2}) \Rightarrow \\
\qquad \textit{pf}\ (\textit{same-topology} \\
\qquad\qquad (\textit{succ}\ \textit{nv1})\ (\textit{succ}\ \textit{ne1})\ \textit{nf1} \\
\qquad\qquad \textit{nv2}\ \textit{ne2}\ \textit{nf2}) \\
\textit{same-topology-e1-f1} :: \textit{nv1}{:}\textit{num} \Rightarrow \textit{ne1}{:}\textit{num} \Rightarrow \textit{nf1}{:}\textit{num} \Rightarrow \\
\qquad \textit{nv2}{:}\textit{num} \Rightarrow \textit{ne2}{:}\textit{num} \Rightarrow \textit{nf2}{:}\textit{num} \Rightarrow \\
\qquad \textit{pf}\ (\textit{same-topology} \\
\qquad\qquad \textit{nv1}\ \textit{ne1}\ \textit{nf1} \\
\qquad\qquad \textit{nv2}\ \textit{ne2}\ \textit{nf2}) \Rightarrow \\
\qquad \textit{pf}\ (\textit{same-topology} \\
\qquad\qquad \textit{nv1}\ (\textit{succ}\ \textit{ne1})\ (\textit{succ}\ \textit{nf1}) \\
\qquad\qquad \textit{nv2}\ \textit{ne2}\ \textit{nf2}) \\
\textit{same-topology-v1-f2} :: \textit{nv1}{:}\textit{num} \Rightarrow \textit{ne1}{:}\textit{num} \Rightarrow \textit{nf1}{:}\textit{num} \Rightarrow \\
\qquad \textit{nv2}{:}\textit{num} \Rightarrow \textit{ne2}{:}\textit{num} \Rightarrow \textit{nf2}{:}\textit{num} \Rightarrow \\
\qquad \textit{pf}\ (\textit{same-topology} \\
\qquad\qquad \textit{nv1}\ \textit{ne1}\ \textit{nf1} \\
\qquad\qquad \textit{nv2}\ \textit{ne2}\ \textit{nf2}) \Rightarrow \\
\qquad \textit{pf}\ (\textit{same-topology} \\
\qquad\qquad (\textit{succ}\ \textit{nv1})\ \textit{ne1}\ \textit{nf1} \\
\qquad\qquad \textit{nv2}\ \textit{ne2}\ (\textit{succ}\ \textit{nf2}))
\end{array}
$$

LF signatures for topology equivalence judgements (continued)

References

[1] T. Altenkirch. Integrated verification in type theory. Lecture notes for a course at ESSLLI 96, Prague, 1996.

[2] L. Augustsson. Cayenne – a language with dependent types. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1998.

[3] Bruce G. Baumgart. Winged edge polyhedron representation. Memo AIM-179, Stanford University, October 1972.

[4] T. Dey, H. Edelsbrunner, S. Guha, and D. Nekhayev. Topology preserving edge contraction. Technical Report RGI-Tech-98-018, Raindrop Geomagic Inc., 1998.

[5] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[6] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. Technical Report TR 93-01-01, University of Washington, January 1993.

[7] W. S. Massey. *Algebraic Topology: An Introduction*. Graduate Texts in Mathematics. Springer, 1977.

[8] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programing*, 14(1):69–111, 2004.

[9] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[10] Frank Pfenning. *Computation and Deduction*, chapter 3, pages 37–87. unpublished, 2001.

[11] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer-Verlag, 1999.

[12] A. Stump. Imperative LF meta-programming. In C. Schürmann, editor, *4th International Workshop on Logical Frameworks and Meta-Languages*, 2004.

[13] Luiz Velho. Mesh simplification using four-face clusters. In *SMI '01: Proceedings of the International Conference on Shape Modeling & Applications*, page 200. IEEE Computer Society, 2001.

[14] Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2005. under review.

[15] H. Xi. Facilitating program verification with dependent types. In *Proceedings of the International Conference on Software Engineering and Formal Methods*, pages 72–81, 2003.

Vita

Joel R. Brandt

Date of Birth April 30, 1983

Place of Birth Cedar Rapids, IA

Degrees B.S. Summa Cum Laude, Computer Science, May 2005
Washington University in St. Louis, St. Louis, Missouri

M.S. Computer Science, May 2005
Washington University in St. Louis, St. Louis, Missouri

Publications Joel Brandt. Run-time Modification of the Class Hierarchy in a Live Java Development Environment. Senior thesis, advised by Kenneth Goldman. Technical Report WUCSE-2004-71, Washington University in St. Louis, 2004.

H. Rohrs, B. Bloom, D. Asher, J. Brandt, M. Reinders, R. Chhatwal, P. Berger, and W.R. Gentry. Inexpensive Mass Spectrometer for Field Applications. In *4th Workshop on Harsh Environment Mass Spectrometry*, October 2003.

May 2005