A GRAPH TRANSFORMATION AND VISUALIZATION FRAMEWORK

A GRAPH TRANSFORMATION AND VISUALIZATION FRAMEWORK

By

SCOTT WEST, B.ENG.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (2008) McMaster University
(Computer Science) Hamilton, Ontario

TITLE: A Graph Transformation and Visualization Framework
AUTHOR: Scott West, B.Eng. (McMaster University)
SUPERVISOR: Dr. Wolfram Kahl
NUMBER OF PAGES: vii, 66

Abstract

The study of graph transformation has applications in many fields. The development of visual, interactive tools that operate on graphs is a subject which has received relatively little attention in the area of functional programming. This thesis presents a framework for developing, visualizing, and interacting with graphs in the pure functional programming language Haskell. Additionally, an embedded domain specific language is developed for the purposes of graph transformation within the framework.

Acknowledgements

I would like to firstly thank my family who have always supported and encouraged me in the most important ways throughout my life.

Also, I am also very thankful to Dr. Wolfram Kahl for enabling and encouraging me to pursue graduate studies. His comments and ideas are always informative and they greatly helped shape my work.

Lastly I would like to thank my friends who helped me stay balanced during the stressful times.

Contents

List of Figures

Chapter 1

Introduction

Graphs, as a model, are used and re-used many times in computer science. They offer both a functional and visual model for describing many problems. Some models of computation are represented as graphs, such as Petri nets. However, graphs may also be used to describe less mathematically intense concepts, such as UML class-diagrams. In academia and in practice, there is a need to employ these existing concepts, and also model entirely new problems with graphs.

It is clear that very often graphs represent working models of a problem, and often these models need to be updated and refined. Consider constructing a class diagram in UML, refactoring and other modifying operations continually take place, transforming the underlying graph. This is one instance of graph transformation.

1.1 Goals

The current number of graph transformation frameworks for functional languages is exceedingly small, even smaller when narrowing to those including visualizations of the graphs. As such, a framework for visualizing graphs in a functional-language setting is proposed in this thesis. A portion of this work is an adaptation of existing frameworks in object-oriented settings. This is a novel approach to implementing a graph visualization in the functional style.

Supplementing this visualization is an embedded domain specific language (EDSL) which is designed for expressing graph transformations. This language has the power to replicate existing graph transformation approaches. This allows for familiar concepts to be used by those who are already acquainted with graph transformation theory.

Additionally, the above two concepts should work together to create an interactive, visual, graph transformation system. This provides the developers with the ability to visualize, and thus more easily comprehend the behaviour of graph

transformations that may be under heavy development; in other words, a type of graph-development-sandbox.

Lastly, the entire framework should be very expressive. As the problems and models represented by graphs can be very diverse, the ability to represent as many of these as possible is a distinct advantage.

To summarize, the contribution of this thesis aims to fulfill the following goals:

- Creating a painless visualization of graphs.
- Constructing an expressive embedded domain specific language (EDSL) in Haskell to build graph transformations.
- Linking the above two in a way that allows for easy extension to different kinds of graphs.

1.2 Organisation

This thesis is organized into several chapters. The initial background, found in Chapter 2 outlines the existing visualization and GUI libraries in Haskell. Additionally, this section also provides the definition for graphs which forms the basis of many of the concepts found in the rest of the document.

The visualization aspect of the framework is modeled after an existing object-oriented design, [Joh92]. We provide a modification/simplification of the OOHaskell approach [KL05] in Chapter 3 to facilitate the translation from an object-oriented design into a functional language such as Haskell. The implementation of the visualization is not provided in this thesis, as it is immense. The overall design of the visualization, using the object-oriented approach, is given in Chapter 4.

Also, a graph transformation EDSL is given in Chapter 5. It provides a collection of combinators to express graph transformations in an intuitive and comprehensible way. We show that our EDSL can be used to implement the single-pushout approaches in Chapter 6.

A bridge between the visualization and the transformation is also given. The ability to allow very non-functionally-styled code to coexist and work with more native Haskell code is very important. As such we present a method of integrating the two approaches in Chapter 7. This integration allows the Haskell developer to work in a familiar style, while still leveraging the power of an interactive visualization. A succinct definition of the working model is given in Section 7.1 and the high-level GUI is given in Section 7.2 as literate Haskell code. Finally we have future work and a few concluding remarks in Chapter 8.

Chapter 2

Background and Related Work

In this chapter, we see the explanation of the fundamental concepts which will form the basis of the work to come. Canvas frameworks are covered in Section 2.1; Section 2.2 reviews object-oriented frameworks in functional settings; our definition of a graph may be found in Section 2.3, and finally in Section 2.4, we see other visual graph transformation frameworks.

2.1 Canvas Frameworks

We define a **canvas** as a drawing area where the items that are drawn keep some information about themselves, and can be rearranged later. This is in contrast to pure drawing canvases which allow things to be drawn to them, but this is like drawing on paper: what is drawn cannot be rearranged at some point in the future.

There are many existing "canvas" frameworks. These canvas frameworks do not tackle the general problem of graphs, but they do provide a base on which to build one. These include:

- SOE – Used in Hudak's textbook [Hud00], this graphics interface provides only limited functionality.
- wxHaskell – A binding of wxWidgets to Haskell. wxWidgets is a cross-platform C++ library which utilizes the native widgets of the underlying platform.
- qtHaskell – Multi-platform C++ GUI toolkit rebound to Haskell. The Qt implementation includes a built-in canvas, though not provided by the Haskell binding yet.
- Gtk2Hs – Gtk, "GIMP Tool-Kit", multi-platform C GUI toolkit bound to Haskell. No canvas is included, but drawing primitives are available via the Cairo drawing system.

Both Gtk2Hs and and wxHaskell are fairly mature platforms, and qtHaskell is still very new. These toolkits are all more-or-less direct translations of their underlying imperative counterparts. Generally, they have very little in the way of a functional programming approach to user interfaces.

Such functional style GUI frameworks do exist however. They often provide a combinator approach to composition of GUIs, and model the interactions between components in a more functional style. Such Haskell frameworks include:

- Fudgets – A completely original Haskell widget system, unmaintained. [CH93]
- FranTk [Sag99] – A binding to the Tk GUI toolkit, using an early functional reactive programming (FRP) framework, Fran [EH97]. Currently unmaintained.
- Fruit – Widgets based on the arrow FRP approach [CE01].
- TV – "Tangible Values" combinators to combine visualization and control of values [Ell07].
- FieldTrip – A preliminary OpenGL based library for building animated and interactive 3D geometry. `http://haskell.org/haskellwiki/FieldTrip`.
- Reactive – A more recent implementation of functional reactive programming (FRP) concepts. `http://haskell.org/haskellwiki/reactive`.

2.2 Object-Oriented or Functional?

As an early design decision, an object-oriented (OO) style was chosen to be employed for the graphical "canvas" portion of the visualization. This was to take advantage of the already numerous OO canvas designs. In particular, we used HotDraw [Joh92] as a starting-point for our design. As the design of the canvas itself is not the main focus of this thesis, this was found to be an acceptable deviation from the functional style of program design.

There has been some work done already on integrating object-oriented concepts into Haskell. These approaches have ranged from library- to extension-level implementations. A library-level implementation does not use any language extensions, where other approaches require the language to be modified to provide the necessary object framework.

One library-level approach is OOHaskell [KL05]. It employs type-level programming to implement a special record syntax applicable to OO programming. It allows concepts such as interfaces and abstract classes to be represented, by virtue of the type-level programming that is employed.

Another avenue that was not pursued is to extend the Haskell language itself, such as in O'Haskell [Nor02]. Here, Haskell is extended with sub-typing and monadic objects. The combination of the two allows for inheritance and also "objects" as a stateful data-type.

Further comparison and review of these approaches will be provided in Chapter 3.

2.3 Graphs

Here give some background on the fundamental concepts underlying graph transformations. We define the working graph definition that we use for the remainder of the thesis, as well as an introduction to category theory which is important for understanding Chapter 5 and Chapter 6.

Definition 2.3.1. A **graph** is a tuple $(N, E, L, src, trg, lab_N, lab_E)$, where:

- N is the possibly infinite set of nodes.
- E – the possibly infinite set of edges.
- L – the possibly infinite set of labels.
- $src : E \rightarrow N$ is a total function mapping an edge to its source node.
- $trg : E \rightarrow N$ – a total function mapping an edge to its target node.
- $lab_N : N \rightarrow L$ – a labelling function to associate labels with nodes.
- $lab_E : E \rightarrow L$ – a labelling function to associate labels with edges. □

Graphs are useful in many contexts, including computation. Term graph rewriting is one such way to model computation. In this case, the graphs represent terms, and transformation of term-graphs represent steps of the computation. In fact, this technique is used in [PJL92] to evaluate expressions (represented as graphs) to model the execution of a functional programming language.

2.4 Graph Editing and Transformation Frameworks

Although the field is somewhat small, there are examples of graph editing and transformation frameworks. DiaGen [MV95] is a Java-based diagram editing generator. It offers the developer a way to produce graph editors that work on arbitrary kinds of

graphs, similar to what is offered in this thesis. DiaGen is based on hyper-graphs, and the underlying transformation model uses hyper-graph grammars. The framework also provides "direct manipulation" of the contents of the graph, allowing the diagram to be directly modified by the user in an interactive manner.

The concepts in Chapter 6 are used in existing graph rewriting systems such as AGG [Tae99]. AGG is an attributed graph transformation system. It is based on the single pushout approach to graph transformation with negative application conditions (NACs). It is a visual environment in which the productions are specified graphically. The implementation is in Java, and the attribute types must be valid Java types. The interface provides sets of graph transformations to select, and bears some similarity to traditional UML diagram editors.

Chapter 3

A Middle Road to OO in Haskell

Here we give an overview of existing OO techniques, and also develop a modification to an existing technique for our needs. Firstly we give a brief justification for the dismissal of O'Haskell [Nor02] in Section 3.1. In Section 3.2 we discuss our experiences with the OOHaskell approach, and motivate a slight compromise in elegance, prompted by increased maintainability and efficiency. We then briefly outline the idioms we do use in our approach, namely a particular setup of existential types (Section 3.3), and record-object creation (Section 3.4) in tandem with a generalised monad and reference setup. We then give a method of creating the objects (Section 3.5) and go over a few flaws present in our technique (Section 3.6).

3.1 The O'Haskell Option

There have already been a couple attempts at embedding OO methods into the Haskell programming language. They include the OOHaskell library [KL05] which is covered in the next section, and the O'Haskell extension to the Hugs Haskell interpreter [Nor02]. O'Haskell added the concept of monadic objects to simulate stateful objects in Haskell. Monads are gone over in more detail in Section 5.2. Additionally, the idea of subtyping was introduced, making inheritence feasible.

The objects in O'Haskell are introduced by making a "template". This template is essentially a class-level description of an object. They create objects in what is called the "Cmd" monad, from this template. This is very similar to the tactic that OOHaskell and we use.

The O'Haskell method was examined for appropriateness, though considered to be unsuitable due to maintenance reasons. Proper maintenence is crucial when selecting a piece of software that will form an integral part of the base of a project. The maintenence issue is even more important because O'Haskell is implemented as an extension to the Haskell language. This means that only interpreters and compilers

supporting the extension will be able to understand it. The use of an extension may be warranted if it is also widely supported. However, O'Haskell is a generally unaccepted extension, not supported in any of the commonly used Haskell compilers.

The combination of the reasons mentioned above make the use of O'Haskell as our underlying object-oriented structure ill-advised.

3.2 OOHaskell Review

The OOHaskell library is a piece of software utilizing type-level programming to supply extensible records in Haskell, a feature which is currently lacking in the Haskell '98 standard [PJ+03]. It builds on the HList work [KLS04] to do this. Also there is a overview of techniques using mutable state, inheritance, and other object-oriented concepts.

3.2.1 Building Records

As mentioned, the HList library offers the ability to build extensible records. Extensible records are very important to an object-oriented approach because they provide essential structure to bind functions to. Also because they are extensible, one may update or add more to them, as is the case with inheritance and overriding in OO. For this, the HList approach uses heterogeneous lists to build its records. The basic elements of these lists are:

$$
\begin{array}{ll}
\mathit{emptyRecord} & :: \mathit{Record}\ \mathit{HNil} \\
(\ .*.\ ) & :: \mathit{HExtend}\ e\ l\ l' \Rightarrow e \rightarrow l \rightarrow l' \\
(\ .=.\ ) & :: l \rightarrow v \rightarrow F\ l\ v
\end{array}
$$

Where the above represent the empty record, record construction, and creation of label-value fields. F here denotes the type of a field. For example, one could use the above to build a record

```
record1 :: Class1 → IO Class1
record1 self = return (
      a .=. 3
   .*. b .=. "Some string"
   .*. emptyRecord
                        )
```

The interpretation of the above is that this record has two fields. One field can be accessed with label a, and the other with b. These labels refer to an *Integer* and

a *String*, respectively. They are appended together using the .*. operator, and is terminated with the *emptyRecord* type. Compared to existing Haskell syntax, this is similar to the usage of the list constructor (:) and the empty list ([]).

Note that *a* and *b* from the *record1* definition need not have any values of their type. As a rule, they should never have their values evaluated, so they can be constructed using an "undefined" value. Also, since their role is specifically as labels, they are wrapped in a *Proxy* type to help make the purpose of the type as a label clear. A label may be something like

```
data A = A
a :: Proxy A
a = proxy
```

The HList-type records, being extensible, allow for easily adding more fields. Since these records are glorified lists, one can add more fields by simply appending, using .*. . Additionally, updates can be performed using the . < . operator. l .=. v . < . r will update the value at the already-existing label l with the value v in the record r. Other operators are also made available for related operations such as union.

In OOHaskell, one is able to use inheritance by first instatiating the super class, then appending more records to it. If we wanted to take our *record1* from above and extend it, it would look something like:

```
record2 self = do
  super ← record1 self
  return (
        c .=. True
    .*. d .=. [1,2,3]
    .*. super
        )
```

The above example constructs a *record1* first and names the resulting record *super*. This *super* then forms the base of the record we're about to create. We can add methods and override methods using the previously mentioned * and . < . operators.

The self argument we see in the record definitions is a convenience so that one may be able to call methods of the class from within the not-yet-constructed methods of that class. This is similar to the usage of the *this* keyword in other object oriented languages.

3.2.2 Up-/Down-casting

Once the basic extensible records are available, OOHaskell then uses them to build up more complicated object-oriented structures. Using the extensible records, one is

also able to modify the records in complicated ways. For instance, an "interface" is given in the form of

```
type Class1 = Record (
        A :=: Int
  :*: B :=: String
  :*: HNil
                      )
```

Here, the :=: and :*: are type-level equivalents of the field creation (.=.) and record construction (.*.) functions given earlier. Such infix type-operators can be used to more clearly express the types of classes.

If we are given the above interface, we are able to "narrow", or up-cast, records satisfying some permutation of the *Class1*-type into *Class1*. The pre-condition for this to occur is known as record sub-typing. Note that this *narrow* operation is destructive: it prunes the extra fields out of the record. This eliminates the possibility of down-casting later.

An alternative to the destructive up-casting is given, using the *Typeable* type-class. This is run-time typing which allows the original types to be recovered. This information is embedded during the narrowing process, which allows down-casting to occur later. This down-casting operation is safe, wrapping successful casts in *Just*; otherwise *Nothing* is returned.

3.2.3 Experiences with OOHaskell

The ability of OOHaskell to properly replicate the object-oriented style is quite impressive, given that it does not use any esoteric extensions to the Haskell language, as O'Haskell does. The OOHaskell approach puts forth a technique that allows a developer to quickly establish a class-based design. It allows the developer to easily create basic classes, with mutable state using *STRef* or *IORef* references. From here, it is also very straight-forward to extend the basic classes by both overriding and adding new methods to the class.

The ability to treat OO-concepts in a natural way is the main strength of OOHaskell. It allows the user to create nearly a one-to-one connection with an object-oriented design. It is implemented without extending the Haskell language which is also very appealing, though it does make use of some common extensions to the Haskell 98 standard.

However, as the classes grow, and their use increases, it becomes more infeasible to continue using the OOHaskell approach. The type signatures of the classes become harder to define and keep track of. This is caused by the heavy use of

type-classes and their role in tracking pre-conditions of record operations, such as narrowing, lookups, etc.

Additionally, due to the large amount of type-level programming, the compilation performance continually decreases as the use of the OOHaskell records increases. The scale of type-level programming occurring using this method is not typical of the average Haskell program. Therefore, this may be one area that can be made more performant in time, but will only occur as compiler-side optimizations are introduced.

This use of the type-system in Haskell seemed to be the cause of long compile-times in early versions of the framework. It took approximately 20 minutes on a 2GHz machine to compile, and occupied 300MB of memory. Using our less ambitious approach explained later in this chapter, compile times shrink down to one minute on a functionally equivalent code-base. The simplified approach we take does increase boiler-plating slightly though.

3.3 Interface Style Rationale

It seems that OOHaskell, although rejected, provides a number of basic techniques which are desirable to us. Programming to an interface is one such technique, as interfaces can be described in OOHaskell using type signatures.

However, since we cannot leverage the power of extensible records, we must have a different approach. The standard Haskell solution when interfaces are needed is to use type-classes, which are very similar to interfaces.

Type-classes, though, are slightly too rigorous for our uses. Since we are designing a canvas, we need our canvas to be able to hold many different types of figures at the same time. This is achieved by defining the interface exposed by any figure; the natural analogue to Java-like interfaces in Haskell are type-classes. Java-like interface types then correspond to *existential types* [LO94] over the respective type classes.

For example, the interface for figures is defined by the *FigureC* type class, and the existential type *Figure* can contain values of any type f for which an instance of *FigureC* has been defined:

data *Figure* = *forall* $f \circ$ *FigureC* $f \Rightarrow$ *Figure* f

This usage of existential types to separate the interface and implementing type also helps to resolve highly recursive classes. Classes which recursively utilize other classes can use the opaque existential type to avoid module recursion when compiling.

For convenience, after creating such an existential type, one can re-expose the class instance for the underlying type. This is done by defining an instance of the class for the existential type. With the previous example, this would look like:

```
instance FigureC Figure where
  fig_moveBy (Figure f) p = fig_moveBy f p
  ...
```

The existential type now essentially just defers the definition of its member functions to the enclosed type.

We will have several instances of *FigureC*. Such figures range from simple graphical changes (display rectangles, triangles, ellipses, etc), to more complicated instances such as connecting figures, and labelled figures.

3.4 Object Implementation Technique

The previous section gives a method of creating interfaces for the classes in a reasonable way. There is still a need for a technique to create the "skeleton" of the classes. Where OOHaskell had used the type-level extensible records of HList [KLS04], we now substitute regular Haskell records. However, we do sacrifice the brevity of having extensible records. Though, the tail polymorphism outlined in [KL05] does help somewhat in this regard.

A small example may make the approach more clear. First we create a structure to bind our functions to,

```
data Person m ref other = Person
  { getName :: m String
  , setName :: String → m ()
  , getId :: m Int
  , setId :: Int → m ()
  , other :: other
  }
```

The *m* and *ref* are intended to be an instance of the *HasRef m ref* type-class. The *other* parameter allows additional functionality to be added to the object.

The type-class definition for *HasRef* is as follows:

```
class (Typeable1 m, Typeable1 ref, Monad m) ⇒ HasRef m ref where
  newRef :: v → m (ref v)
  readRef :: ref v → m v
  writeRef :: ref v → v → m ()
```

The object-oriented encoding we use relies mainly on two things: having a monad, and having references. The *HasRef* type-class ensures we have these things, if we are given an instance of it. It is a simple type-class which we use to abstract away

the existence of references *ref* for a particular monad *m*. The use of *Typeable1* as super-classes become important when using *interface* in Section 4.2

One natural choice here are the *IO* monad, along with the *IORef* references. Thus we provide a default implementation, *HasRef IO IORef*. Similarily we provide one for *ST s* monads, as they include *STRef*s.

Note that often times uniquely generated identifiers are needed. Haskell provides these in the form of *Unique*, which returns results in the IO monad. For more general usage though, this could be extended by using the *StateT* monad transformer and adding an *Integer* which is incremented as unique values are produced.

Given the above record, we can then extend it using the "tail" portion of the record. Namely, we create another record which supplies further methods, and load it into the tail-section.

Such a record may look like:

```
type SkilledPerson m ref other =
  Person m ref (SkilledPersonDelta m ref other)
data SkilledPersonDelta m ref other = SkilledPersonDelta
  { skill :: m String
  , skill_other :: other
  }
```

We can combine the delta and the *Person* to get a *SkilledPerson*. This is done by simply setting the *other* field of person to contain a *SkilledPersonDelta*. Except for the monad generalisation using *HasRef*, this is the same technique given in the tail-polymorphic mutable-state section given in [KL05]. We have found it very useful for reducing boiler-plating and excessive code duplication.

3.5 Constructors and Variables

As in [KL05], the instance variables of a class are then represented by *IORef*s or *STRef*, depending on the choice of underlying monad either *IO* or *ST*. The *Ref*s are essentially strongly-typed pointers.

We also adopt the OOHaskell "functions as constructors" method. A reasonable approximation of the OOHaskell technique is attained by essentially replacing the specialized .=. operator with the more regular record-assignment =. Also, the concatentation .*. just becomes the field seperator , and because of this we eschew the empty record at the end.

For example, for the *Person* record we gave in the previous section, we could have a constructor as follows:

```
person :: HasRef m ref ⇒
  String → Int → Person m ref other → m (Person m ref other)
person name ident self = do
  nameRef ← newRef name
  identRef ← newRef ident
  return
    (Person
      { getName = readRef nameRef
      , setName = writeRef nameRef
      , getId = readRef identRef
      , setId = writeRef identRef
      }
    )
```

We can see that the initial name and identification number are parameters to the *person* constructor-function. They are subsequently passed to the state-creation function *newRef*, which will allow us to have mutable state for the resulting object.

The as-of-yet unexplained argument is the *self* parameter, of type *Person*. From the name one could guess that its role is similar to the *this* keyword often found in object-oriented languages such as Java and C++. This is the exact same concept as the techinque seen in Section 3.2.

For us, the *self* argument, just like in [KL05], makes it possible to call methods on the not-yet-constructed object that is in the midst of being defined. We could rewrite the *getName* method to use the *self* parameter, appending the ID number to the end of a name.

```
getName = do
  str ← readRef nameRef
  num ← getId self
  return (str ++ "-" ++ show num)
```

To actually generate the object, we must use the monadic fixed-point combinator over the constructor-function. It appears slightly strange to allow some "other" *Person* to occupy the self argument. This is solved by always using the *mfix* function to construct the objects. This is what allows us to have the self-referencing behaviour outlined above.

The monadic fixed-point function, *mfix* [EL02], is typed as follows:

```
mfix :: MonadFix m ⇒ (a → m a) → m a
```

(*MonadFix* is a subclass of *Monad* providing only *mfix*.) In the case of our *Person* class, we can make a new object by supplying the proper initial values, and passing the result to the fixed point combinator.

$personObj \leftarrow mfix\ (person$ `"Jeff"` $12345)$

One can see that in this case *mfix* bears some similarity in usage to the *new* operator in languages such as C++ or Java. The type-name which would usually follow is now followed by a function. Once again, this draws directly from the OOHaskell [KL05] style.

One can merge the interface style given in Section 3.3 with this technique to create larger and more diverse object-oriented programs. This is done by defining instances of the appropriate type-class for record-types. For example, if there was a type-class *PersonClass*, we would create an instance of it for *Person*. The last step is to create an existential type around *PersonClass* to allow multiple differently-typed *PersonClass* instances to coexist.

3.6 Problems and Summary

The main drawback of these techniques is the considerble boiler-plating. In particular, in OOHaskell where one can mainly operate directly on the extensible records, we have replaced these pieces with a type-class, a record to hold the "object", and an existential type, which incidentally needs to be re-instanced as a member of the type-class.

Every time one wants to create a new existential type, it must re-instance itself as a member of the enclosed type-class. This can lead to some length and seldom interesting definitions of the class-members. There may be some hope to alleviate some of this through the use of Template Haskell, first proposed in [SPJ02]. This is so far an unexplored avenue of research, as it is somewhat tangential to the purpose of this thesis.

It is important for users of our framework to understand the above style of object-oriented programming in Haskell. It offers a reasonable degree of object-oriented style while still offering full compatibility with the Haskell language. Also, it is used extensively, if not completely, in the translation of the HotDraw design into its Haskell counterpart. Though, liberties were taken to utilize more functionally-styled programming techniques where possible, such as in method definitions. However, the scaffolding of the entire canvas system is built around the object technique that we have briefly outlined here; thus it is important to comprehend its usage.

Chapter 4

Canvas Overview

The design of the canvas inherits many ideas from the HotDraw design [Joh92] and the JHotDraw implementation [EG98]. This was done to reduce the amount of time taken on this part of the framework, not looking to reinvent the wheel in terms of canvas design.

A high-level view of the design is given in Section 4.1. A functional approach to extending this design is given in Section 4.2. An intermediate representation of the canvas drawing as a graph is given in Section 4.3. Lastly, in Section 4.4 a developer's guide is provided, the API of the most important classes.

4.1 High-level Design

At a high level, the design goes together in the following way:

- *Figure* – the base canvas visual element.
- *Drawing* – collects and manages figures.
- *Drawing View* – a particular view of a drawing.
- *Tool* – acts upon a drawing, view, or figure.

This design can also be examined from the model-view-controller (MVC) point-of-view. The closest approximation to the MVC design pattern is to consider figures and drawings as the model; the drawing view is naturally the view, and the tools are the controller.

Of course there are many more elements to the design. However, knowing these is sufficient to understand the parts of the framework that are relevant for our graph transformation system.

4.1.1 Figure

Figures are the entities on the canvas which can be seen and interacted with. They are the basic elements on the canvas, as places, transitions, tokens, and arcs are the basic elements of a Petri net. Given the graphical nature of the figures, they offer facilities for:

- Drawing themselves; this is done via the Cairo rendering engine [Cai08].
- Storing and updating their dimensions; this includes position and information such as width and height. Currently the interface dictates that this information is represented as a rectangle. More complicated shapes may be supported in the future, for more accurate "collision detection" with other figures.
- Listening interface for other figures, the drawing, etc. For example, an edge must "listen" to the figures it is attached to. Otherwise, it would be unable to redraw itself when its source or target figures changed location or size.
- Transforming state to a string representation, and parse from a string as well. Obviously a requirement for saving and loading figures.
- Generate connection points for connection figures (edges). This allows figures to dictate where they may be connected to by edges.
- Manage what drawing and parent figure it has, if any. Since figures may not exist on a drawing, it is useful to be able to determine if a figure is currently on a drawing or not. Also, figures may be nested inside other figures, if the other figures support it. This means that they will be deleted when their parent is deleted, and moved when their parent moves.
- Create modifying handles to change its dimensions. Handles are essential for modifying figures after they are created. A modification could be resizing the extents of the figure, for example.

The associated types in the framework are given by the *FigureC* type-class and also the *Figure* type. The *Figure* type is existential, admitting only instances of the *FigureC* type-class. In practice we use *Figure* whenever we want to talk about figures, much like we would in an OO language. *FigureC* is used to define the Haskell interface to the type.

4.1.2 Connection Figures

A sub-class of figures are the connection figures. In a graph setting, these model the edges that run between nodes. They are special in that they are the first basic kind

of figure which relies on other figures. They must modify themselves to match their source and target figures' position. Therefore, they are the first instance we have seen of something which must "listen" to a figure's change of state.

In addition, they are also special because figures are generally created in an interactively piece-wise fashion. That is, the source and target are selected separately by the user. Thus, we have to allow connection figures to be incomplete for some amount of time while they are being created. This is accomplished by allowing the end points of the figure to be either concrete connections to figures or just points. Any connection that is not currently being created or edited should have both a source and target connector associated with it.

The following additional functionality is thus added to the base figure:

- Storage and retrieval of possible start and end connectors.
- Organizing a list of intermediate points for edges with many segments.

The relevant type-class and existential type are *ConnFigC* and *ConnFig*, respectively.

4.1.3 Drawing

Now that we have a brief outline of the basic building blocks of the graph, we can start to think about how to put them together. On a very simple level a *Drawing* is just a collection of figures. It is one of the most basic elements of the framework, as its only real additional responsibility is to be queried about what (if any) figure is at some location. It is the object-oriented equivalent of what will later be our graphs. Drawings must be instances of the *DrawingC* type-class and also wrapped in the *Drawing* existential type, as needed.

4.1.4 Drawing View

The drawing itself is not responsible for visualizing itself, it just contains the knowledge of how to draw itself if need be. It is useful to consider the ability to draw itself the knowledge of what to draw, but nowhere to actually draw it. The place where the drawing actually takes place (the "paper") so to speak is the drawing view. A drawing view, like the V of MVC, is one particular view of the drawing.

Therefore, it is not surprising that the drawing view may have some distortions to the original visualization such as translation or scaling. A *GtkDView* is offered by default in the framework, using a Cairo backend. So far only linear operations are supported at the level of Cairo, so doing more complicated transformations would have to wait until Cairo offers such capabilities or the drawing back-end is changed.

Additionally, the drawing view is responsible for "overlay" material. This includes the handles present on some figures, and the selection of figures as well. Any drawing view must be an instance of the *DViewC* type-class and the associated existential type is *DView*.

4.1.5 Tools

Tools are a way to modify the canvas and figures interactively. Tools contain hooks for detecting mouse presses, releases, movement, and dragging. It seems that this is enough versatility to replicate a good number of behaviours that are common in canvas-like applications.

Tools have access to the drawing view, as they operate on the transformed (scaled, translated) view of the drawing presented in the view. This is a necessity, as the tool is a user-facing component of the design and therefore must use the user-facing visualization, and merely accessing the drawing would be insufficient.

The tools which are included by default are the panning and zooming tool, the selection tool (which includes moving figures and handles). Also, there is an included creation tool, which can handle figure creation if the figure requires only a "drag" (to specify the new figure's extents) or "click" (to specify that a default-sized figure should be placed under the cursor) interaction scheme.

The same naming scheme holds for tools as well, with the type-class and existential type being *ToolC* and *Tool*, respectively.

4.2 Extensibility

Once we have the basic elements outlined above, we can start to incorporate them into a style more suitable to functional programming. So far the entire design is assumed to be implemented in an object-oriented style, which we now try to escape from. We must therefore make an abstraction away from the more detailed, object-oriented view of a canvas.

To a functional programmer, a figure is just the on-screen representation of some data-type, such as a string, integer, etc. So, we try to expose that directly through the use of the *FigureLabel* type-class. The *FigureLabel* type-class is a distilled version of the functionality offered by the figure classes. We use a technique of "embedding" a *FigureLabel* instance into the special *LabelledFig* figure. This *LabelledFig* figure is sort of a figure with a hole in it. By this we mean that *LabelledFig* is not a complete definition of a figure, and needs to wrap itself around a *FigureLabel* instance so that it may defer the things it does not know to that instance.

Really, what is desired is the ability to specify figures with varying levels of detail. If only basic changes need to occur, only modifying the way a figure is drawn

(for example moving from a rectangle to an ellipse), then it should be easy and straight-forward to do so. If however, deeper modifications are required, then there should be support to do that as well.

Using the *FigureLabel/LabelledFig* method allows us to partially specify the representation of the data in a succinct way without getting very deep into the internals of how figures are meant to operate. The *FigureLabel* type-class provides an interface to specify how a particular node should be:

- Drawn, according to its current bounding rectangle.
- Interacted with, given that it has been clicked on. This interaction could be as simple as adding a token to a place in a Petri net, or it could spawn a complicated dialog that modifies the figure in complicated ways.
- Dimensioned. The dimension can either be nothing (and thus be changed by the user) or set to some constant value, which cannot be updated by the user by means of resizing handles.
- Parsed and saved as a string.

Thus we have a type-class interface that looks like the following rather direct translation of the above:

```
class (Show a, Typeable a) ⇒ FigureLabel a where
  draw       :: a → Rect → Render ()
  size       :: a → Maybe Point
  parseLabel :: Parser a
  interface  :: HasRef m ref ⇒ LabelFig m ref → DView m ref → IO ()
```

The preceding bit of Haskell tells us that before the *a* type can have an instance of *FigureLabel* we have to first have instances of *Show* and *Typeable* available for it. For simple data-types the Glasgow Haskell Compiler (GHC) can provide these definitions automatically using the **deriving** keyword.

The *draw* function takes the payload of the *LabelledFigC* instance and the size of the figure, and constructs a visualization, which is represented by a *Render* computation.

Definition 4.2.1. The *Maybe* type represents data which may exist, or not. It is defined as:

```
data Maybe a = Just a | Nothing
```

The *size* function allows us to specify some constant size for the figure by returning *Just* if there is a default size. Otherwise *Nothing* is returned and the figure can be resized as the user wishes.

The *Show* super-class is used for saving to files. For loading, a compatible parser *parseLabel* must be provided.

The function *interface* is currently the only way to specify custom interaction; it is invoked when the figure is double-clicked on. It could be used to open a menu or a dialog box, or to directly invoke some useful action on the graph. This interactive behaviour is why we the result of the function is an m computation.

Since the general style we try to employ is not use a particular monad, it is common to see IO as the monad m. However, the purpose of this function is to largely to deal with user interaction, so this is not strange. To deal with the associated typing issues, we have also developed a small group of functions. These functions facilitate the execution of IO actions in the context of *interface* where it must not be assumed that the monad m is in fact IO. This is achieved using the *Typeable1* constraints on *HasRef* (see Section 3.4), which allow *interface* to determine at run-time (and possibly at compile time in case of specialisation) whether m is indeed IO. If the IO action to be performed will be executed within an IO monad, then it is done. Otherwise, a default action is executed in the non-IO monad. This default action is likely to be *return* () a neutral operation. It is for this reason we require the *Typeable1* super-class for m and *ref* in Section 3.4.

The basic drive of the *FigureLabel* type-class and the associated *LabelledFig* figure is to allow simple figures to be created with a minimum amount of programming overhead, as we demonstrate in the next section. Obviously, for more involved figures, the developer would need to delve deeper into the canvas design to create the more complicated figures.

4.3 Intermediate Representation

Given that the ability to define many aspects of the visualization is realized in the previous section, and in Chapter 5 we give a method of transformation and selection from a graph, we do need a representation of that graph.

For this purpose we have the *GV lab m ref* data-type, representing a visual graph with node labels of type *lab*, running in a monad m with references *ref*. As mentioned in Section 3.4, we require availability of a *HasRef m ref* instance.

We give node and edge labels asymmetric treatment here. This is due to the relative ease with which one can implement a hypergraph in which the edges are again nodes, and then the node labels can be partitioned into both hypergraph-node and hypergraph-edge labels. It should be said that this approach does not allow the

same amount of flexibility that having edge labels would have, such as changing line thickness, style, etc.

GV is a combination of Erwig's inductive graph [Erw01] Haskell implementation and the underlying OO drawing, essentially just a tuple of the two. It allows us to operate on the structure of the graph at the same time as we are able to modify the visualization of that graph through the drawing.

Many functions are defined through the *GV* interface that allow the graph to be modified including adding nodes, edges, querying the degree of nodes, and numerous other graph operations. The interface often returns monadic results due to the use of OO drawing objects.

It is important to note that we force ourselves to return new *GV*s if we modify the graph in any way. This becomes important in Chapter 5 where we differentiate between non-modifying and modifying transformations. We restrict the non-modifying transformations to only use *GV* functions, which do not contain a graph-visual result.

4.4 Developer's Guide

4.4.1 Figure Interface

Figures form the basis of all the elements in a drawing. They are combined and connected to produce more complicated and possibly nested graphs.

The *Figure* class itself must also be an instance of the *Object* class. This is to support things like identification of *Figure*s.

class (*Typeable fig*, *Object fig m ref*) ⇒ *FigureC fig* (*m* :: ∗ → ∗) *ref* **where**

- *fig_draw* :: *fig* → *m* (*Render* ())

Create a *Render* computation representing the visual characteristics of the *Figure* Assuming that the value of the *FigureC* is later wrapped into a *Figure*, this will later be composed by the drawing into the larger picture. Coordinates used in this function should be not be relative to the bounding box of the *Figure*, they should be absolute.

- *fig_getBounds* :: *fig* → *m Rect*

Return the bounding box of the *Figure*. The bounding box is represented as a rectangle, even if the *Figure* itself does not have this shape. It is a sort of rough-estimate of the size of the *Figure*.

- *fig_setBounds* :: *fig* → *Rect* → *m* ()

Set the bounding box of the *Figure*. This follows the same reasoning and bounds-semantics as the previous function for getting the bounds. Setting the bounds is most often used by functions which modify the shape of the figure.

- *fig_moveBy* :: *fig* → *Point* → *m* ()

Moves the *Figure* by some amount specified by the *Point* parameter. One effect of this will be to offset the bounding box by the vector (*Point*) specified.

- *fig_addFigureListn* :: *fig* → *FigListn m ref* → *m* ()

Add a *FigListn* to this *Figure*. Events from this *Figure* will be transmitted to all of its *FigListn*s.

- *fig_removeFigureListn* :: *fig* → *FigListn m ref* → *m* ()

Removes a *FigListn* from the *Figure*. This *FigListn* will no longer receive any events from this *Figure*. If the *FigListn* is not currently "attached" to this *Figure* then nothing should happen.

- *fig_clearFigListn* :: *fig* → *m* ()

Removes all *FigListn*s from this *Figure*.

- *fig_createHandles* :: *fig* → *m* [*Handle m ref*]

This function creates a list of *Handle*s which will be able to control the extents of the *Figure*.

- *fig_contains* :: *fig* → *Point* → *m Bool*

Query whether the *Point* supplied lies within this *Figure*. **Note:** *Point*s that lie within the *Figure* may not necessarily lie within the bounding box and vice versa. The latter may be the case for shapes such as circles.

- *fig_rectContains* :: *fig* → *Rect* → *m Bool*

A convenience function to query the *Figure* if it lies completely within the supplied *Rect*.

- *fig_getParent* :: *fig* → *m* (*Maybe* (*Figure m ref*))

Get the parent *Figure* of this *Figure*. If this *Figure* does have a parent, it will be wrapped in a *Just* constructor, if not, the *Nothing* value will be returned.

- *fig_setParent* :: *fig* → *Figure m ref* → *m* ()

Set the paren to this *Figure*.

- *fig_getDrawing* :: *fig* → *m* (*Maybe* (*Drawing m ref*))

Get the owning *Drawing* of this *Figure*. *Figures* may be currently in a *Drawing*, or not. *Figures* with a *Drawing* associated with them will return a *Just d* value, where *d* is the *Drawing*. Unassociated *Figures* will return *Nothing*.

- *fig_setDrawing* :: *fig* → *Maybe* (*Drawing m ref*) → *m* ()

Sets the associated *Drawing*. If the drawing is to be set, the value should be wrapped in a *Just*. Otherwise, if the *Figure* is to be unassociated from a *Drawing*, then a value of *Nothing* should be passed.

- *fig_setLayer* :: *fig* → *Int* → *m* ()

Set the drawing layer that this *Figure* should be presented on. The lowest (first drawn) layers have the lowest numbers. A good number to start at would be 0. A typical interpretation of this in terms of composite figures would be a "nesting" level.

- *fig_getLayer* :: *fig* → *m Int*

Retrieve the current drawing layer of the *Figure*.

- *fig_findConnector* :: *fig* → *Point* → *m* (*Maybe* (*Connect m ref*))

Take a candidate point, and produce a connection (*Connect*) which will bridge this figure and an edge. If no suitable connection can be made, then *Nothing* is returned. This may happen if the candidate point is outside the *Figure*.

- *fig_click* :: *fig* → *MouseEvent* → *DView m ref* → *m* ()

A way to embed *Figure*-specific behaviour. This function is called when the *Figure* is double-clicked on. For example, this could be a pop-up dialog (requiring *IO*) or a perhaps just an internal change to the figure requiring no additional user-input (*m* may be *ST s* here for instance).

- *fig_delete* :: *fig* → *m* ()

A way to notify the figure that it has been removed from a drawing. This gives the *Figure* a chance to perform any final actions.

- *fig_read* :: *fig* → *FigureParser m ref*

This function is a parser which can create a *Figure* of the same type as this one. This is to account for differences in *Figure* specific details. For example, a rectangular figure would likely store width and height, where-as a ciruclar figure would store radius. This is used in the loading process for *Drawings*.

- *fig_show* :: *fig* → *m String*

Returns a *String* representation of this *Figure* to be used during the saving process of *Drawings*.

FigureParser is mentioned above. We define it here as a *Parser* which can compute a *Figure*, the *Figure*'s reference, and also a list of associated *Figures*.

type *FigureParser m ref* = *Parser* (*m* (*Figure m ref*), *Int*, [*Int*])

We see *m* emerge here due to the fact that internally the objects are using *ref*s as references.

The associated *Figures* could be children *Figures* for composite-*Figures*, or they could be start and end *Figures* for *ConnFigs*.

4.4.2 Connection Figure Interface

ConnFigs (connection figures) must be an instance of the *Figure* class. Additionally, since their visualization and existence is dependent on their source and target *Figures*, they also must be an instance of the *FigListn* (figure listener) class.

class (*FigListnC cfig m ref*, *FigureC cfig m ref*) ⇒
ConnFigC cfig m ref **where**

- *cfig_setStartConn* :: *cfig* → *Maybe* (*Connect m ref*) → *m* ()

Set the starting (*Connect m ref*) of the connection figure. You can make the starting connection "free" by specifying *Nothing* here as a parameter instead of *Just*.

- *cfig_getStartConn* :: *cfig* → *m* (*Maybe* (*Connect m ref*))

Get the (*Connect m ref*) for the start of the connection figure. A *Just* value means that a starting connector exists, *Nothing* otherwise.

- *cfig_setEndConn* :: *cfig* → *Maybe* (*Connect m ref*) → *m* ()

Analogous to the *cfig_setStartConn* function except for the ending connector.

- *cfig_getEndConn* :: *cfig* → *m* (*Maybe* (*Connect m ref*))

Analagous to the *cfig_getStartConn* function except for the ending connector.

- *cfig_setStartPt* :: *cfig* → *Point* → *m* ()

Lacking a starting connector to provide the *Figure* with a starting point, this function will supply a starting point for the *ConnFig*. This would be used for example, during *ConnFig* creation, when actual start and end connectors are not necessarily available. Though this interface could merge with *cfig_setStartConn* and demand a *Either Point* (*Connect m ref*) to set either the start as a point or connection, it was done this way to provide more informative function naming. One can discern the required type by the name of the function quite easily.

- *cfig_getStartPt* :: *cfig* → *m Point*

This retrieves the starting point. If a starting *Connect* is specified, it is used to obtain the point. Otherwise this should be the point that was set with *cfig_setStartPt*.

- *cfig_setEndPt* :: *cfig* → *Point* → *m* ()

Analogous function to *cfig_setStartPt*.

- *cfig_getEndPt* :: *cfig* → *m Point*

Analogous function to *cfig_getStartPt*.

- *cfig_addNode* :: *cfig* → *Point* → *m* ()

Adds a node to the start of the list of nodes (*Points*).

- *cfig_getNodes* :: *cfig* → *m* [*Point*]

Get the entire list of nodes which make up this *ConnFig*.

- *cfig_setNode* :: *cfig* → *Int* → *Point* → *m* ()

Set a particular entry in the list of nodes.

- *cfig_update* :: *cfig* → *m* ()

Update the node to be aligned with the proper positions on the *Drawing*. This is likely to be called in cases where the start or end *Figure* have been updated, and now we have to realign this *ConnFig* to match their movement.

Lastly, two functions which are useful when dealing with *ConnFig*s are given. Note that both are derived from the definition of the *ConnFig* interface, and thus will work on any instance of *ConnFig*, and do not have to be redefined.

- *getStartFig* :: *ConnFigC cfig m ref* ⇒ *cfig* → *m* (*Maybe* (*Figure m ref*))
  getStartFig cf =
  cfig_getStartConn cf ≫=
  maybe (*return Nothing*) (*liftM Just* ∘ *cn_getOwner*)

Gets the starting *Figure* of this connection based on the starting connector.

- *getEndFig* :: *ConnFigC cfig m ref* ⇒ *cfig* → *m* (*Maybe* (*Figure m ref*))
  getEndFig cf =
  cfig_getEndConn cf ≫=
  maybe (*return Nothing*) (*liftM Just* ∘ *cn_getOwner*)

Gets the ending *Figure* of this connection based on the ending connector.

4.4.3 Drawing Interface

A *DrawingC* instance is firstly an *Object*. Secondly, it is a *FigListn* (figure listener) due to its requirement to redraw itself when the *Figure*s it contains change. Lastly, it is a *FigureContainer*, for obvious reasons outlined in the previous design chapter.

class (*Typeable drw*, *Object drw m ref*,
FigListnC drw m ref, *FigureContainer drw m ref*) ⇒
DrawingC drw m ref **where**

- *draw_draw* :: *drw* → *m* (*Render* ())

A function which produces a *Render* computation, embodying cumulative rendering of all the *Figures* which this drawing contains. This also means drawing things in the proper order, according to layer.

- *draw_clone_map* :: *drw* →
  m (*drw*,
  Map (*Figure m ref*) (*Figure m ref*),
  Map (*ConnFig m ref*) (*ConnFig m ref*)
)

A way to "clone" or copy a *Drawing*. The new *Drawing* is returned, along with mappings of the *Figures* and *ConnFigs* in the old *Drawing* to the ones in the newly created *Drawing*.

- *draw_contains* :: *drw* → *Figure m ref* → *m Bool*

A convenience function to query whether the *Drawing* contains the *Figure* supplied as a parameter.

- *draw_findFigure* :: *drw* → *Point* → *m* [*Figure m ref*]

This function allows all *Figures* which fall under the given point to be returned.

- *draw_findFigureWithout* :: *drw* → *Figure m ref* →
  Point → *m* [*Figure m ref*]

Similar to the previous function, but excludes the *Figure* given. This is for situations when it is useful to find something during *Figure* creation, which is not the *Figure* that is being created. For example, when creating a *ConnFig*: to find a start or end *Figure* one has to exclude the *ConnFig* that is being created from consideration, as it cannot be its own start/end *Figure*.

- *draw_findFiguresWithin* :: *drw* → *Rect* → *m* [*Figure m ref*]

Find all the *Figures* which lie completely within a given rectangle.

- *draw_addDrawListn* :: *drw* → *DrawListn m ref* → *m* ()

Add a *DrawListn* (drawing listener) to this drawing. It will be notified of drawing events.

- *draw_remDrawListn* :: *drw* → *DrawListn m ref* → *m* ()

Remove a *DrawListn* from this drawing. Events will no longer be sent to this listener.

4.5 Implementation Details

Visualization concerns are taken care of by a combination of object-oriented techniques, adapted to a functional style, and by way of the Gtk2Hs binding to obtain widgets. More specifically the Cairo drawing library is employed to attain a PostScript-like syntax of drawing. This affords us the luxury of being able to render not only to the screen, but also to PostScript, and PDF document formats directly, as well as SVG file types.

Chapter 5

Graph Transformation Environment

The graph selection and transformation environments are designed to be instances of the Haskell *Monad* and *MonadPlus* type-classes. The purpose of these environments is to hide the representation of the graph, and to allow complex computations to be built from a limited number of low-level combinators.

5.1 Category Theory Background

As categories and category theory are the basis for many concepts in computer science, it is not surprising to see their appearance in graph transformations as well.

Definition 5.1.1. A **category** C can be defined as in [Pie91]:

- A collection of objects $Obj(C)$.
- A collection of arrows, or morphisms, between objects. If an arrow f exists between objects A and B, we write $f : A \rightarrow B$.
- Operations *dom* and *cod* denoting the domain and co-domain of an arrow. For example, if $f : A \rightarrow B$, then $dom(f) = A$ and $cod(f) = B$. The collection of all arrows which have domain A and co-domain B are represented by $C(A, B)$.
- A binary operator $\circ$, composition. For any two arrows which share a common domain and co-domain, composition is defined on those arrows. If $f : B \rightarrow C$ and $g : A \rightarrow B$ then $f \circ g : A \rightarrow C$. Some may also define a notational variant of the composition operation, ;, which merely reverses the order of the arguments of $\circ$.

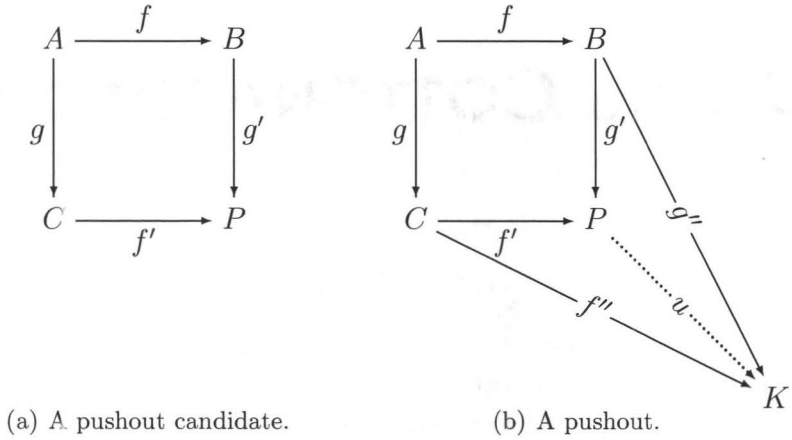
- Every object A has an arrow id_A which is the identity arrow. This arrow is a left and right identity for composition. That is to say that $f \circ id_A = f = id_B \circ f$ for any f. □

Because the definition is seemingly innocuous, there are many concepts which are accurately modeled as a category. For example, **Set** is a category in which all objects are sets, and all morphisms are total functions between sets. We can see immediately that the identity arrow on each set is merely the identity function and that arrow composition is function composition.

For our purposes, the following definition holds some interest:

Definition 5.1.2. A **pushout** (also called a **pushout completion**) along some morphisms $f : A \to B$ and $g : A \to C$ is a triple (P, f', g') such that the diagram in Figure 5.1(a) commutes, and for all other (K, f'', g'') such that Figure 5.1(b) commutes, u is the unique morphism from P to K.

Definition 5.1.3. If a category C has pushouts for all objects A, B, D, and morphisms $f : A \to B$ and $g : A \to D$, we say that C *has pushouts.*



(a) A pushout candidate. (b) A pushout.

Figure 5.1: Pushouts.

A pushout can be seen as some kind of generalized union. Another useful way to think about a pushout is if B and C represent two things that have common pieces stored in A, then constructing the pushout means that the information in A is used to glue B and C together.

Now, as one might guess, categories themselves can be objects in their own category-category. In this case, the arrows between the objects in this category are called functors.

Definition 5.1.4. A **functor** is an arrow between objects in the category of categories. If C and D are categories, then a functor $F : C \rightarrow D$ is a morphism from objects of C to objects of D. If A is an object of C then $F(A)$ denotes the corresponding object in D.

Additionally, if $f : A \rightarrow B$ is an arrow in C, then the corresponding arrow in D is denoted $F(f) : F(A) \rightarrow F(B)$.

Lastly, the following must hold:

- For all objects A in C, $F(id_A) = id_{F(A)}$.
- For all composable arrows f, g, $F(f \circ g) = F(f) \circ F(g)$.

5.2 Haskell Monad Primer

In category theory, as defined in [ML71], monads are:

Definition 5.2.1. A **monad** on a category C is:

- an endofunctor $T : C \rightarrow C$.
- a "unit" natural transformation, $\eta : Id_C \rightarrow T$.
- a "join" natural transformation, $\mu : T^2 \rightarrow T$.

Such that both Figures 5.2(a) and 5.2(b) commute. The notation ηT represents $\eta_{T(X)}$, while $T\eta$ represents $T(\eta X)$.

From [Pie91] a natural transformation is defined:

Definition 5.2.2. Let C and D be categories, and let F and G be functors from C to D. A **natural transformation** η from F to G is a function that assigns to every C-object A, a D-arrow, $\eta_A : F(A) \rightarrow G(A)$ such that for any C-arrow $f : A \rightarrow B$,

$$G(f) \circ \eta_A = F(f) \circ \eta_B$$

To see how the definition of a monad relates to computation, it may be informative to look at the Haskell definition of a *Monad*.

In Haskell, the *Monad* type-class is often used to define computation. One of the most easily understandable kinds of computation is the *Maybe* monad. It embodies computations that can fail.

For example, if one were to write a slope function for a line, it may return *Just* 2 as an answer or possibly *Nothing* if the line is vertical.

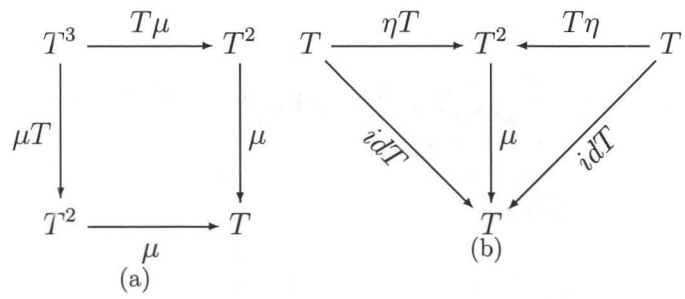


Figure 5.2: Commuting monad diagrams.

Haskell has a fairly strong basis in category theory, and as such many of the concepts are explained using definitions found in category theory. One such definition is that of a functor. Remember that a monad includes a functor, so it is reasonable to assume that Haskell monads can replicate the idea of a functor in some way.

Definition 5.2.3. The *Functor* type-class makes sure that the arrow-mapping property of categorical functors is maintained. The object mapping is guaranteed by Haskell's type system.

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

A simple definition for a Maybe-instance of Functor could be:

```
instance Functor Maybe where
  fmap f (Just a) = Just (f a)
  fmap f Nothing = Nothing
```

This makes a new function that will take values out of *Just*, apply the argument function f, and then re-wrap the result in *Just* again. If the input is *Nothing* however, we return *Nothing*. This is consistent with the definition of functors however, as *Nothing* is a value of the *Maybe* type.

With the definition of *Functor* we can start to look at what makes a *Monad*:

Definition 5.2.4. The *Monad* type-class is defined in the following way,

```
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
```

The definition of the *return* function for *Maybe* has an obvious choice: the *Just* data-constructor. By its very definition it takes any type and wraps it in a *Maybe*. The *return* operation also corresponds directly to the "unit" from functors, η.

The infix operator $\gg=$, or *bind*, does not have a direct translation from the categorical definition of monads. Looking to the type signature for $\gg=$ above, it is possible to gain some intuitive understanding of the bind operator.

$m \gg= f$ essentially says that a computation m is performed with some result, that result is then used as the input to the following function f, which results in another computation.

In the case of *Maybe* the definition for bind is as follows:

$$(Just\ v) \gg= f = f\ v$$
$$Nothing \gg= f = Nothing$$

As before, we see that a *Nothing* value gives us no way to construct any more values. If a value is found however, then we use that as the input to the next computation and let that resulting computation be the result of f.

We notice here that we have no requirement of *Functor* anywhere here. That is because we can define *fmap* given a *Monad m*. We can define *fmap* as follows:

$$fmap :: Monad\ m \Rightarrow (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$$
$$fmap\ f\ ma = ma \gg= return \circ f$$

This shows that in some sense *Monad* implicitly includes the *Functor* type-class. In effect, the η natural transformation compacts two layers of computation together. Given this, we can show that the definitions of η and $\gg=$ are isomorphic.

$$\begin{aligned} m \gg= f &= \eta((T(f))m) \\ \eta(m) &= m \gg= id \end{aligned}$$

5.3 Backtracking Monads

Since we would like a computation environment that provides as much expressiveness as possible, so-called *backtracking monads* are used. Backtracking monads give a way for computations to be defined with some kind of unwinding of the computation when a failure occurs. For an overview of backtracking monad transformers, including a treatment of fair choice and its application to logical programming, see [KSFS05].

Haskell will actually handle most of this for us through the definition of the *MonadPlus* type-class.

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

The *MonadPlus* definition adds the concept of failure, *mzero*, and union of computations, *mplus*. If we assume that the computations expressed in the *m a* type have multiple values, this gives us a backtracking monad.

To understand how this works more fully, consider the following Haskell code:

```
evenInt :: [Integer]
evenInt = do
  n ← [1..3]
  guard (even n)
  return n
```

Here, the monad is the list monad, []. Values of *n* are selected from the the list containing numbers one to three. If the guard predicate fails, and the number is not even, the monad will backtrack to the previous line and other values from the list will be taken and tested. If the predicate is true, then we continue on and return the even *n*.

One useful primitive seen above is conditional failure, or *guard*. The definition of guard is informative and simple, so we give it here:

```
guard :: MonadPlus m ⇒ Bool → m ()
guard True = return ()
guard False = mzero
```

5.4 Selection Monad Definition

The first monad we present is the *SelectM* monad.

```
data SelectM lab m ref v = SelectM (GV lab m ref → m [v])
```

A *SelectM* type signature shows not only the result of the computation, but also the type of the node labels. Therefore, having *SelectM String IO IORef Integer* is a selection on a graph containing *String* nodes, and the computation itself returns an *Integer*. The monad in which the computation takes place is the *IO* monad, using *IORef*s for references.

A *SelectM* is really just a function that will take the underlying graph and produce a list of results. As mentioned before, the list here is to facilitate backtracking behaviour.

The effect is analogous to the application of monad transformers in [LHJ95] and used in [LH96]. In particular it is sort of like a manual application of a *ListT* and *EnvT* monad transformers to an m monad. Essentially this is a backtracking reader monad as mentioned by Jones in [Jon95].

The *GV* type from Section 4.3 is the argument to the function wrapped in the *SelectM* data-type constructor. This allows us to query the graph through the *GV* interface. We only create *SelectM* computations using *GV* functions which do *not* return a new *GV* as a result. This enforces the intended reader monad behaviour of the computation.

We can then make this *SelectM* type an instance of the *Monad* type-class as follows:

```
instance Monad (SelectM lab m ref) where
  SelectM gf >>= f =
    SelectM $ λg →
      gf g >>= liftM concat ∘ mapM (($g) ∘ unSelectM ∘ f)
  return x =
    SelectM (λ_ → return [x])
```

Although it is a reasonably dense definition, it essentially says this:

- $\gg=$ – run the first transformation (gf) on the input (g). *unSelectM* unwraps the function that is contained in the *SelectM* data constructor. This will produce a list of results. Take that list of results, and take the f and run it over each member (again, producing another list of results). Now we have a list-of-lists of results, so the last step is to concatenate them together (*liftM concat*).

- *return* – this function essentially returns a constant one-element list containing x.

Without becoming too formal, we can see that the implementation satisfies the monad laws.

- Left unit, $return\ a \gg= f \equiv f\ a$ – The result a is passed as a singleton list into the *mapM* function, which of course then will produce a singleton result, which is then concatenated. So in fact the list of results that comes out of $f\ a$ is the only result.

- Right unit, $m \gg= return \equiv m$ – By similar reasoning as previous, the result of the computation m is unmodified by the *return* function. It merely wraps it in a singleton list again, which is immediately thrown away again by use of the *liftM concat* function.

- Associativity, $(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)$ – Since the *SelectM* monad definition for $\gg\!=$ essentially uses only function composition and mapping, it is reasonable to conclude that the associativity of $\gg\!=$ holds.

Adding the instance for *MonadPlus* is much easier to understand:

```
instance MonadPlus (SelectM lab m ref) where
  mzero = SelectM (λ_ → return [])
  mplus (SelectM f) (SelectM g) =
    SelectM (
      λx → do
        r1 <- f x
        r2 <- g x
        return (r1 ++ r2)
          )
```

Failure is defined as the function always returning the empty list of results. Adding two computations together is also very straight-forward in that we first compute the results for the individual computations, then simply concatenate the results together.

For example, one could use *SelectM* to select two distinct nodes from a graph:

```
get2nodes = do
  n1 ← sel_node
  n2 ← sel_node
  guard (n1 /= n2)
  return (n1, n2)
```

5.5 Selection Operations

We use a function in the previous example called *sel_node*. The Selection module includes a wide variety of primitive operations which can be used to build more complicated selections. The Selection module does not export the *SelectM* constructor we see in the previous section. This is to try to help ensure that there are no side effects occuring. Also, we try to design our exported functions so that their implementations also have no side effects. This effectively makes the *SelectM* a true reader monad.

The initial functions that are provided by the framework are as follows;

- *sel_node* :: *SelectM lab m ref* (*VNode lab*)

  returns some node from the graph; via backtracking we may get any node from here.

- *sel_edge* :: *SelectM lab m ref* (*VEdge lab*)
  returns some edge from the graph

- $a \twoheadrightarrow b$ :: *SelectM lab m ref* (*VNode lab*) →
  SelectM lab m ref (*VNode lab*) →
  SelectM lab m ref (*Vnode lab*, *VEdge lab*, *VNode lab*)

  "`a ~~>> b`" – returns a triple (s, e, t) where s is source-node, e is the connecting edge, and t is the target node. a and b are computations each returning a node, backtracking.

- *sel_val* :: *VNode lab* → *SelectM lab m ref* (*Maybe lab*)
  returns the label of a node n, non-backtracking.

- *sel_in* :: *VNode lab* → *SelectM lab m ref* [*VEdge lab*]
  returns the incoming edges to n, non-backtracking.

- *sel_out* :: *VNode lab* → *SelectM lab m ref* [*VEdge lab*]
  returns the outgoing edges from n, similar to *sel_in*.

- *sel_src* :: *VEdge lab* → *SelectM lab m ref* (*VNode lab*)
  returns the source node of e.

- *sel_trg* :: *VEdge lab* → *SelectM lab m ref* (*VNode lab*)
  returns the target node of e.

- *sel_scc* :: *SelectM lab m ref* [*VNode lab*]
  returns a list of strongly connected components from the underlying graph.

- *sel_results* :: *SelectM lab m ref v* → *GV lab m ref* → *m* (*Maybe v*)
  return a result of the selection. *Nothing* if no results.

- *sel_resultList* :: *SelectM lab m ref v* → *GV lab m ref* → *m* [*v*]
  return the list of results from the selection.

The general design goal of the Selection module is to create a "little language" to perform a very dedicated task. This is not unlike the *Little Theories* method presented in [FGT92]. The drive is to permit a similar style as to that in the Parsec parser combinators [LM01]: primitive operations allowing the expression of useful computations.

Given the limited set of above basic operations, we are able to define more complicated computations very easily. The resulting definitions are also very understandable. To show how one may build such selections, we may define *onlyNode*. *onlyNode* will return only nodes that match satisfy some predicate, filtering out the rest.

```
onlyNode :: (p → Bool) → SelectM p (VNode p)
onlyNode p = do
  n ← sel_node
  guard (p n)
  return n
```

5.6 Transformation Environment

The transformation environment complements the selection environment. Where the selection monad allows us to query the graph about features and calculate results based on these, the transformation environment *GraT* allows us to directly augment and modify the underlying graph.

This is vastly different from the selection environment where it is impossible to change the structure of the graph. The transformation environment is strictly more powerful than the selection environment, as it is able to do everything the selection environment can and more.

```
data GraT lab m ref v =
  GraT ((BigMap lab m ref, GV lab m ref) →
    m [(BigMap lab m ref, GV lab m ref, v)])
```

The *GraT* monad could have also almost been described in terms of a backtracking *StateT* [LHJ95] transformed *m* monad, with *BigMap* and *GV* as state. However, this becomes impossible as we must maintain some internal properties. One is that we copy the *GV* on every operation. This is to ensure that we can still backtrack in the face of changes to the underlying drawing, which would normally make backtracking not work. Since this produces a new *GV*, and thus new references for the nodes and edges, we must also add the homomorphism from the old to the new *GV* to the *BigMap* state.

The *GraT* data-type is fundamentally different from the *SelectM*. Firstly, we can see that it has two extra results, and an extra input.

```
type BigMap lab m ref = BigMap
  (Map (VNode lab m ref) (VNode lab m ref))
  (Map (VEdge lab m ref) (VEdge lab m ref))
```

The extra input/output pair of *BigMaps* are partial homomorphisms used to establish connections between the underlying graph (the GV value seen as another input/output pair) before and after the operation. As we can see, *BigMap* is really two maps, one for the nodes, and one for the edges. Lastly, we have the result of the computation, the v type in the list of results.

```
instance Monad (GraT lab m ref) where
  GraT s >>= f =
    GraT (
      λ(b, g) → do
        (g', b') ← gv_copy g
        gr_res ← s (compMap b' b, g')
        sels ← mapM (λ(b'', g'', r) → (unGraT (f r)) (b'', g'')) gr_res
        return (concat sels)
          )
  return x = GraT (λ(b, g) → return [(b, g, x)])
```

The $\gg=$ operator has some significant differences from the *SelectM* definition. Firstly, we see that we use the *gv_copy* function to copy the input graph. This function returns a copy of the graph, as well as a mapping so we can translate nodes and edges from the pre-copy graph to the post-copy graph. This is the purpose of the *BigMap* mentioned earlier.

Now that we have a copy of the input, we run that through the left-hand $GraT$, and save the results. Then, we perform the right-hand function on the list of results, in the *mapM* line. We obtain now a list of lists of results, which we concatenate together and return.

The *return* definition is much more straight-forward, simply not modifying the mapping or the underlying graph while returning the argument value, x.

The left and right unit monad laws hold as before for the *SelectM* monad. However, the associativity of $\gg=$ is not as clear as it was before. Since we have now copy the underlying graph, and also must take care to compose the resulting maps. However, assuming these operations associate properly (map composition does) the $GraT$ monad is also associative in $\gg=$.

As stated above, the $GraT$ computation is also able to backtrack, very much like the *SelectM* computation. However, unlike the *Monad* instance, there are practically no differences between the $GraT$ and *SelectM MonadPlus* instance:

```
instance MonadPlus (GraT lab m ref) where
  mzero = GraT (λ_ → return [])
  mplus (GraT f) (GraT g) =
    GraT (
```

```
λx → do
  r1 ← f x
  r2 ← g x
  return (r1 ++ r2)
    )
```

The *GraT* monad is more powerful than the *SelectM*. It can do everything that the *SelectM* monad can, and we actually allow the developer to inject *SelectM* in to the *GraT* monad using the *selToGra* function. This enables us to integrate the selection and transformation monads, performing a selection then passing the results to the transformation.

This means also that we can recycle all of the existing *SelectM* functions into the *GraT* computation when we need them, eliminating the need to define them twice.

The *GraT* module does contain some other functions though, which are useful for modifying the underlying graph.

These include:

- *addNode* :: *FigureLabel lab* ⇒ *lab* → *GraT lab m ref* (*VNode lab m ref*)
  add and return a new node to the graph with the *FigureLabel* value *a*.

- *connect* :: *VNode lab m ref* → *VNode lab m ref* →
  GraT lab m ref (*VEdge lab m ref*)

  connect the nodes *s* and *t*, returning the resulting edge.

- *deleteNode* :: *VNode lab* → *GraT lab m ref* ()
  delete a node, and also the incident edges.

- *deleteEdge* :: *VEdge lab* → *GraT lab m ref* ()
  delete an edge.

- *runTransform* :: *GraT lab m ref v* → *GV lab m ref* →
  m (*Maybe* (*GV lab m ref*)) |

  runs a *GraT* computation on the supplied *GV* graph. The resulting (new) graph is returned, and *Nothing* is returned on failure.

The above functions provide the basic tools with which one can create transformations. They form the basic tools that can be used to create more complicated combinators.

The interface to the GraT module, as the Selection module above, does not export the data constructor. Also, it is not permitted to allow the *BigMap* homomorphisms to become publically available. Though the restriction on modification is lifted, compared to the *SelectM* monad, we still must be careful allowing arbitrary computations m into the monad. Thus, the proper way to interact with the *GraT* monad is through its exported access functions. One may execute a *GraT* computation by using the *runTransform* function.

Chapter 6

Replicating SPO

The single- and double-pushout approaches to graph transformation are a formulation that has already been established. In the following section we first give a brief overview of the pushout approaches. Then, to help show the legitimacy and versatility of the approach in Chapter 5 we also offer an implementation of the pushout approaches translated from our own method.

6.1 SPO Background

In graph transformation theory, there are several ways to model how graph transformations should be performed. Two of them are the *single pushout (SPO)* approach and the *double pushout (DPO)* approach. As may be expected, these pushouts occur in the graph category where objects are graphs, and arrows are partial graph morphisms, as stated in [Roz97]. In particular, for the single-pushout approach, we reproduce the definitions from [EHK+97], in this section for convenience.

Definition 6.1.1. A **production** $p : L \xrightarrow{r} R$ consists of a **production name** p, an injective partial graph morphism r, or **production morphism**. L and R are graphs, and referred to as the left- and right-hand-side (LHS and RHS respectively) of p.

The partiality of r indicates that deletions can occur in the production.

Once we have a production, we can then start to think about which graphs this production can be applied to. This links directly with the LHS of the production, and the property that it must have a *match* in the candidate graph to be transformed.

Definition 6.1.2. A **match** m is a total function from the LHS of a production p to a graph G.

So given a production $p : L \xrightarrow{r} R$ and match m for graph G, if we can find a pushout completion, then we can apply the production to G and obtain our target graph H.

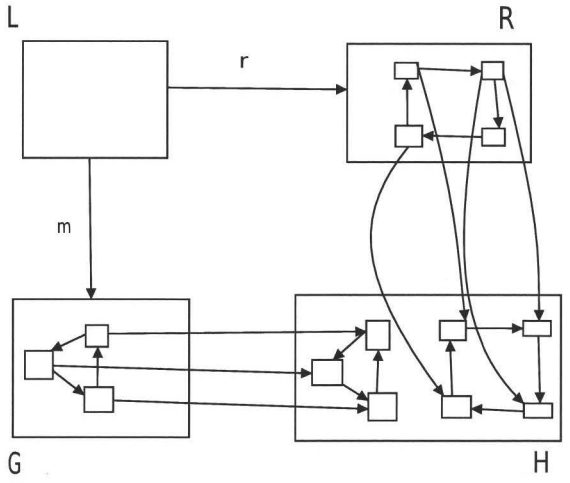


Figure 6.1: Example of SPO with no conditions on matching

We can motivate the use of pushouts by looking back to our general understanding of pushouts as generalized unions. The LHS object is the structure common to both the source graph and the RHS of the rule. So essentially, once we have the match, we can take our source graph at the nodes specified in the LHS, and transform them to be the RHS. This then gives us our target graph (the source graph after transformation).

Contrast Figures 6.1 and 6.2, examples taken from [Kah01], to see the difference in pushout completions when there are some common elements in the match and production morphism.

There are some cases where the behaviour of the derivation needs to be examined. These cases involve conflicts between the production morphism and the match. A production morphism may implicitly specify deletion by leaving some nodes or edges out of its domain. However, the match is total, and therefore must identify these deletions in the source graph. Consider the case of Figure 6.3(a). The a node is simultaneously preserved and deleted at the same time. The result is that the node is deleted, due to the fact that we first identify items using the match, then we derive the resulting graph, meaning that deletion happens second.

Secondly, we may delete a node which is either the source or target of an edge in the source graph, as in Figure 6.3(b). The "dangling edge" that is apparent is also deleted by the same reasoning as the previous case.

Respectively, these two conditions are defined as:

Definition 6.1.3. Give a match $m : L \rightarrow G$ and a production $r : L \rightarrow R$, m is

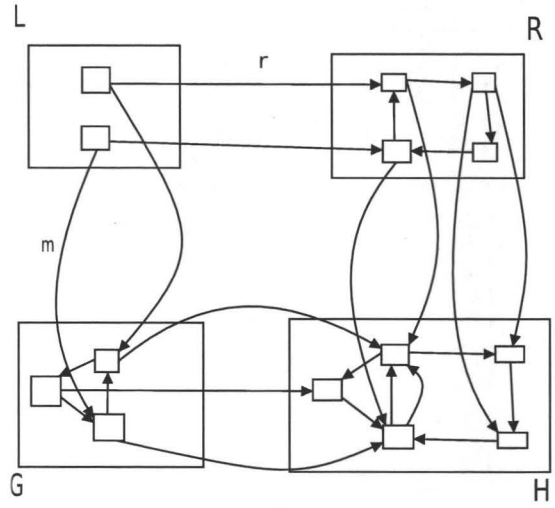


Figure 6.2: Example of SPO with some identification of nodes

d-injective if $m(x) = m(y)$ implies that $x, y \in dom(r)$ or $x, y \notin dom(r)$.

Definition 6.1.4. Give a match $m : L \to G$ and a production $r : L \to R$, m is **d-complete** if for each edge $e \in G_E$ with $source^G(e) \in m_V(L_V - dom(r)_V)$ or $target^G(e) \in m_V(L_V - dom(r)_V)$ we have $e \in m_E(L_E - dom(r)_E)$.

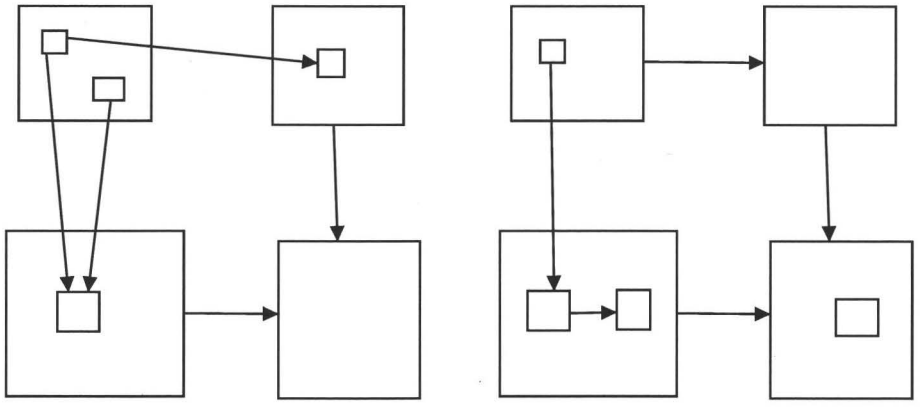
Lastly, the absence of any such confusions is defined.

Definition 6.1.5. Give a match $m : L \to G$ and a production $r : L \to R$, m is **conflict-free** if for every $m(x) = m(y) \implies x, y \notin dom(r)$ or $x, y \in dom(r)$.

6.2 Creating a Match

The matching module produces a matching morphism from a source graph into a target graph. The homomorphism produced is then returned from in a *SelectM* monad. This approach allows additional conditions on the match to be added after, as may be required to build more specific matchings.pay

We define a *Match* as a function which takes a labelled graph, GV, and produces a *SelectM* computation that contains a morphism. This *GraphMorph* represents a morphism between the argument graph, and the implicit underlying graph of the *SelectM* computation.



(a) A deletion conflict. (b) Implicit edge deletion

Figure 6.3: SPO deletions.

```
type Match lab m ref = GV lab m ref →
  SelectM lab m ref (GraphMorph lab m ref)
```

Constructing the SPO matching morphism is done by branching out from some node. This kick-starts the matching process.

```
match :: HasRef m ref ⇒ Match lab m ref
match lhs = case gv_nodes lhs of
  n : _ → matchNode (GraphMorph M.empty M.empty) n lhs
  _ → return (GraphMorph M.empty M.empty)
```

Matching a single node in fact leads to matching all the outgoing edges from that node. Once all reachable edges and nodes have been matched, we pick another node and branch out on that. Of course, we must stop at some point, this is handled by the guard – the condition states that if the domain of the match morphism is contains all the elements of the source graph or if we've already matched the candidate node, then we are done.

```
matchNode :: HasRef m ref ⇒
  GraphMorph lab m ref → VNode lab m ref → Match lab m ref
matchNode partMatch n lhs
  | nonTotalMatch lhs partMatch ∨ n `M.notMember` nodeMap =
    do
      n' ← sel_node
```

```
      let
        newMatch = partMatch{ nmor = M.insert n n' nodeMap }
        outEdges = gv_out lhs n
      foldedMatch ← foldM (matchEdge lhs n) newMatch outEdges
      let remaining = gv_nodes lhs \\ M.keys (nmor foldedMatch)
      case remaining of
        [] → return foldedMatch
        n'' : _ → matchNode foldedMatch n'' lhs
  | otherwise = return partMatch
  where
    nodeMap = nmor partMatch
```

The guard condition, *nonTotalMatch* is false when the target nodes of the morphism cover all nodes in the graph. Another way to look at this is it is non-total when the set of keys of the node map are not equal to the set of nodes in the LHS of the production.

```
nonTotalMatch :: HasRef m ref ⇒
  GV lab m ref → GraphMorph lab m ref → Bool
nonTotalMatch lhs (GraphMorph nodeMor _) =
  fromList (M.keys nodeMor) ≢ fromList (gv_nodes lhs)
```

The edge matching has a precondition that the source of the edge must already exist in the mapping given.

The recursive call in this computation means that we branch outwards, adding nodes and edges as we go.

```
matchEdge :: HasRef m ref ⇒
  GV lab m ref → VNode lab m ref →
  GraphMorph lab m ref → VEdge lab m ref →
  SelectM lab m ref (GraphMorph lab m ref)
matchEdge lhs src partialMatch e = do
  let trg = gv_trg lhs e
  match' ← matchNode partialMatch trg lhs
  let Just matchTrg = M.lookup trg (nmor match')
      Just matchSrc = M.lookup src (nmor match')
  e' ← sel_out1 matchSrc
  src' ← sel_src e'
  trg' ← sel_trg e'
  guard $ src' ≡ matchSrc ∧ trg' ≡ matchTrg
  return $ match'{ emor = M.insert e e' (emor match') }
```

6.3 Implementing the SPO Approach

We model our Haskell-production as close to the categorical definition as we can. The *Production* structure contains a right-hand-side, a left-hand-side and also a graph morphism.

```
data Production lab m ref
  = Production{
    prod_lhs :: GV lab m ref,
    prod_rhs :: GV lab m ref,
    prod_mor :: GraphMorph lab m ref
    }
```

Deletion is modelled by the following function. It is able to determine which nodes and edges have been specified to be deleted. Additionally, it deals with the d-injective and d-complete deletion conflicts.

So-called "dangling" edges are dealt with by the behaviour of the *ConnFig* figures underlying the edges. They delete themselves when one of their source or target nodes are deleted.

When matched edges or nodes are simultaneously preserved and deleted, the deletion takes priority. This is modelled through the order of operations, first the nodes are matched, then they are deleted. Thus, deletion will occur to both, after they have both been identified to the same node in the graph that is to be transformed.

```
spoDelete :: (HasRef m ref, MonadFix m,
  UniqueMonad m, FigureLabel lab) ⇒
  GraphMorph lab m ref → Production lab m ref → GraT lab m ref ()
spoDelete matchMor (Production lhs _ rule) = do
  let delNodesLHS = gv_nodes lhs \\ M.keys (nmor rule)
    delEdgesLHS = gv_edges lhs \\ M.keys (emor rule)
    delNodes = map ((nmor matchMor)M.!) delNodesLHS
    delEdges = map ((emor matchMor)M.!) delEdgesLHS
  mapM_ deleteEdge delEdges
  mapM_ deleteNode delNodes
```

To avoid double-adding, we insert all nodes first, creating a map of the RHS nodes to the new nodes created in the graph.

```
spoAddNode :: (HasRef m ref, MonadFix m,
  UniqueMonad m, FigureLabel lab) ⇒
  Map (VNode lab m ref) (VNode lab m ref) → VNode lab m ref →
  GraT lab m ref (Map (VNode lab m ref) (VNode lab m ref))
```

```
spoAddNode m vn = do
  l ← selToGra $ sel_node_load vn
  n ← addNode l
  let m' = M.insert vn n m
  return m'
```

This map is then given to the edge-adding transformation, which uses it to look up between which new-nodes it should construct an edge.

```
spoAddEdge :: (HasRef m ref, UniqueMonad m,
  MonadFix m, FigureLabel lab) ⇒
  Map (VNode lab m ref) (VNode lab m ref) →
  GV lab m ref → VEdge lab m ref → GraT lab m ref ()
spoAddEdge m rhs ve = do
  let src = gv_src rhs ve
    trg = gv_trg rhs ve
    Just src' = M.lookup src m
    Just trg' = M.lookup trg m
  connect src' trg'
  return ()
```

Insertions are then defined by the elements of the RHS which are not associated with any element in the LHS of the production.

The *spoAddNode* function creates the new nodes and returns the map linking the nodes in the RHS with the new nodes created in the graph. The only remaining task is to then take all candidate edges for insertion and add them.

```
spoInsert :: (HasRef m ref, FigureLabel lab, MonadFix m,
  UniqueMonad m) ⇒
  Production lab m ref → GraT lab m ref ()
spoInsert (Production _ rhs rule) = do
  let insNodes = gv_nodes rhs \\ M.elems (nmor rule)
    insEdges = gv_edges rhs \\ M.elems (emor rule)
  m ← foldM spoAddNode M.empty insNodes
  mapM_ (spoAddEdge m rhs) insEdges
  return ()
```

We put these pieces together in the *spoTransform* computation. The construction is very similar to the SPO approach: first the matching is performed, then the transformation takes place.

```
spoTransform :: (HasRef m ref, FigureLabel lab, MonadFix m,
  UniqueMonad m) ⇒
```

```
    Production lab m ref → GraT lab m ref ()
spoTransform prod@(Production lhs _ _) = do
  m ← selToGra $ match lhs
  spoDelete m prod
  spoInsert prod
```

The above translation from our EDSL into the more formalized SPO approach shows several things. Firstly, it shows that it is possible to construct the more well-known approaches using our EDSL, which is useful for those who already have existing transformations that are specified in terms of the single-pushout approach. Secondly, it is a reasonably short and readable definition. This is an important attribute as one of the qualities we wish to provide is clarity of expression.

Chapter 7

Petri Net Example

In this section, we give a working example of how the framework can be used to build a graph editor. The editor works with Petri nets as the underlying model. The next two subsections are actual literate Haskell source files which are the complete definition for the editor.

Section 7.1 details the model for the graph editor. This includes the visualization of the places and transitions, and also the behaviour of the graph transformation. Additionally, it is contains the necessary definitions for the graph to be loaded and saved.

In Section 7.2 the main GUI is given. Pulling together all of the graphical interface elements, the main-file is responsible for what the user sees surrounding the graph display.

Lastly, Section 7.3 features a discussion of the experience of using the framework. In particular the separation of functional and object-oriented aspects is reviewed.

7.1 Petri Net Label

The purpose of our graph transformation and visualization framework is to facilitate the creation of graph editors. The framework should be able to create not just one graph editor, but a whole class of editors. The goal is to strip away the underlying details, and leaving only the functional description of the visualization and transformations to be defined. Petri nets have already been studied in the context of formalized graph transformations [BCM05], so this seems like a useful area to set a working example.

The code contained below defines the functions which are used by the editor to transform and visualize the Petri nets. It is rather compact, and much of the

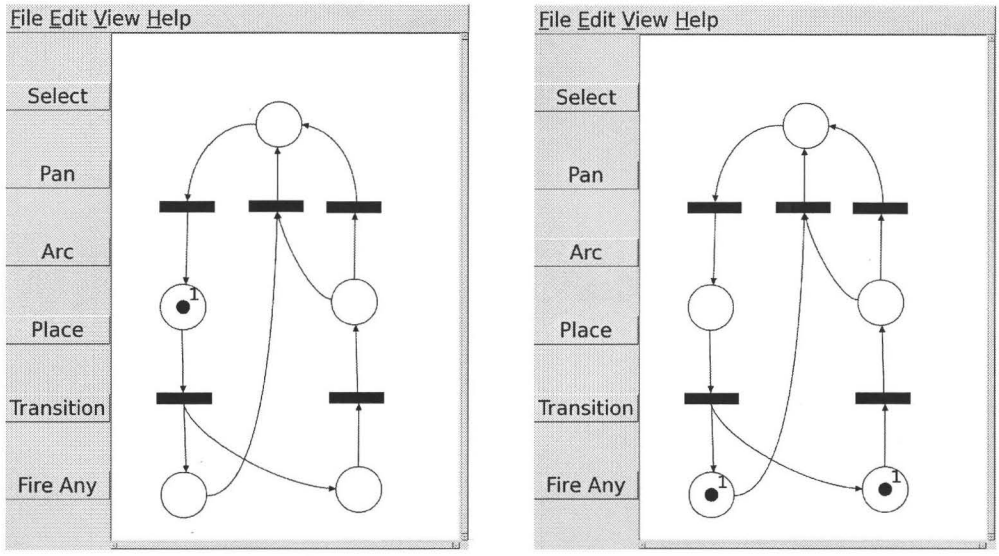


Figure 7.1: Petri net editor and animator resulting from the Petri module.

verboseness comes in the drawing of the figures, rather than in the length of the transformation descriptions.

Although slightly more code is required to create the full editor with the appropriate buttons, and initialize menus, this work is mostly boiler-plating.

The purpose of this example is to show how a fairly reasonable editor and transformation system can be created in very few lines of code. Another purpose is to introduce the idea of the back-tracking selection and transformation environments used to model various operations on the graph in an elegant manner.

The definition of a Petri net is done node-wise, in the sense that we provide the content of each of the nodes (places and transitions), and the framework fills the graph structure.

$$\begin{aligned}&\textbf{data}\ \mathit{Petri} = \mathit{Trans}\\ &\qquad\qquad \mid \mathit{Place}\{\,\mathit{numTokens} :: \mathit{Int}\,\}\ \textbf{deriving}\ (\mathit{Typeable}, \mathit{Show})\end{aligned}$$

We define node label classification predicates:

$$\begin{aligned}&\mathit{isTrans}, \mathit{isPlace} :: \mathit{Petri} \to \mathit{Bool}\\ &\mathit{isTrans}\ \ \mathit{Trans} = \mathit{True}\\ &\mathit{isTrans}\ \ \_ = \mathit{False}\\ &\mathit{isPlace}\quad = \neg \circ \mathit{isTrans}\end{aligned}$$

We first show how the "firing of transitions" can be implemented as a simple transformation of the Petri net using the *SelectM* selection monad and the *GraT* transformation monad.

In the *SelectM* type signature of the transition search function *trans*, the first parameter *Petri* indicates the payload of the nodes in the underlying graph of the selection. The return type *VNode Petri* indicates that the result of the selection computation is a *visual node* with payload of type *Petri* — this can typically be used as anchor for further selections. Other return types, e.g. *Bool* or *Integer*, could be used to extract orreponding kinds of information from the underlying graph.

```
trans :: HasRef s r ⇒ SelectM Petri s r (VNode Petri s r)
trans = onlyNode isTrans
```

The *onlyNode* function receives a predicate and produces a computation that will select a node that satisfies the predicate from the graph. This is a so called backtracking computation, so if at some point the computation after this selection fails, it will back-track to this selection and pick another node that satisfies the predicate.

The selection of appropriate source and target places is then accomplished using the following parameterised *SelectM* computation, which considers the argument transition *tr* as a hyper-edge, and succeeds if all sources of *tr* are non-empty places, and all of the its targets are places. In the case of success, the two place lists are returned in a pair.

The *hyp_src* and *hyp_trg* functions below merely return the list of source and target nodes that each hyper-edge is connected to. This is part of some on-going work to define a hyper-graph specific set of functions.

```
firingSel :: HasRef m r ⇒
  VNode Petri m r →
  SelectM Petri m r ([VNode Petri m r], [VNode Petri m r])
firingSel tr = do
  srcs ← hyp_src tr
  sel_all_guard validSrcPlace srcs
  trgs ← hyp_trg tr
  sel_all_guard isPlace trgs
  return (srcs, trgs)
```

Actually firing a transition is done in the more powerful *GraT* monad by decrementing the token count on all source places and incrementing the counts on the target places. This method of firing even works when a place is connected to a transition more than once.

```
moveToken :: HasRef m r ⇒
  [VNode Petri m r] → [VNode Petri m r] → GraT Petri m r ()
```

```
moveToken ss ts = do
  mapM_ (updNode decToken) ss
  mapM_ (updNode incToken) ts
```

Finally, the full transformation is composed by first lifting the extraction of the sources and targets from the selection monad into the transformation monad, and then applying the token movement computation:

```
tokenTrans :: HasRef m r ⇒ VNode Petri m r → GraT Petri m r ()
tokenTrans tr = do
  (s, t) ← selToGra (firingSel tr)
  moveToken s t
```

To facilitate the above selection computations, we provide a few simple predicates on the Petri net items.

The emptyiness of the places can be determined with the following predicates, which are essential for determining if a particular place is a valid candidate for firing a token.

```
notEmptyPlace :: Petri → Bool
notEmptyPlace = ¬ ∘ emptyPlace


emptyPlace :: Petri → Bool
emptyPlace (Place 0) = True
emptyPlace _         = False
```

A valid source place is then a place that is also non-empty.

```
validSrcPlace :: Petri → Bool
validSrcPlace p = isPlace p ∧ notEmptyPlace p
```

These update functions are to perform the basic operations on the places. They are able to increase the number of tokens, or decrease them depending on if the place is non-empty.

```
decToken, incToken :: Petri → Maybe Petri
decToken (Place 0) = Nothing
decToken (Place i) = Just $ Place (i − 1)
decToken Trans = Nothing


incToken (Place i) = Just $ Place (i + 1)
incToken Trans = Nothing
```

The following defines an instance of the *FigureLabel* type-class. An instance of this class is necessary so the editing framework knows how to display, save, load, and, in some limited sense, interact with the graph.

Since the places and transitions are not nested, and are unlikely to be grown or shrunk, we define constant sizes. This means that they are restricted to these sizes in the visualization, although they will still behave as expected under zooming conditions.

```
instance FigureLabel Petri where
  draw          = drawPetri
  size Trans    = Just $ Point 60 12
  size (Place _) = Just $ Point 50 50
  parseLabel    = parsePetri   -- compatible with Show instance
  interface     = runPredTrans isTrans tokenTrans
```

Connections can only be established from between transitions and places (in either direction), but not between places, nor between transitions.

```
petriCompatible :: Petri → Petri → Bool
petriCompatible Trans (Place _) = True
petriCompatible (Place _) Trans = True
petriCompatible _ _ = False
```

This function will be passed to an adapted *ConnFig* combinator for the arrow drawing "tool", which will modify the stock implementation of the connection figure. This enforces the bipartiteness of the Petri net graph.

Parsing a *Petri* string is just just trying to parse a *Place* or, if that fails, a *Trans*.

```
parsePetri, parsePlace, parseTrans :: Parser Petri
parsePetri = try parsePlace <|> parseTrans
parsePlace = do
  string "Place "
  i ← try (decimal haskell) <|>
    (do
      string "{numTokens = "
      i ← decimal haskell
      string "}"
      return i
    )
  return $ Place $ fromIntegral i
parseTrans = string "Trans" ≫ return Trans
```

This parsing information, combined with the derived *Show* instance, automatically allows Petri net drawings to be saved and loaded with the standard framework load/save functionality.

Drawing is defined using the Cairo interface of Gtk2Hs, which very close in flavour to PostScript programming. The places are drawn as empty circle outlines if there are no tokens present. If there is a non-zero number of tokens, then a token symbol (a filled in circle) is drawn within the place, with a number to the upper right side, to indicate how many tokens are present.

```
drawPetri :: Petri → Rect → Render ()
drawPetri (Place i) r = do
  save
  setSourceRGB 0 0 0
  drawEllipseInRect r
  stroke
  if i ≢ 0
    then do
      let w = rectWidth r * 0.3
      let h = rectHeight r * 0.3
      let Point cx cy = rectCenter r
      let tokenRect =
        Rect
          (Point (cx − w * 0.5) (cy − h * 0.5))
          (Point (cx + w * 0.5) (cy + h * 0.5))
      drawEllipseInRect tokenRect
      fill
      textScaleToRect (show i) (rectShift (Point w (−h)) tokenRect)
      stroke
    else return ()
  restore
```

The transitions are simply a filled-in rectangle.

```
drawPetri Trans r = do
  save
  setSourceRGB 0 0 0
  drawRect r
  fill
  restore
```

As one can see, the size of the Petri net-specific code is small, and highly functional. Adapting the use of monads to embody graph selection and transformation allows elegant expression of graph operations.

7.2 Petri Net Main-file

This file is the main-file of the Petri net editor. Where the Petri.lhs file contained the definitions of specific transformation, saving, loading, and visualizations, this module pulls them together into an application.

```
module Main where
```

The following imports are essential for the creation of the editor. Many of them will likely reappear in most applications developed within this framework.

```
import Control.Monad
import Control.Monad.Fix
```

The Gtk widget and Glade libraries.

```
import Graphics.UI.Gtk
import Graphics.UI.Gtk.Glade
import System.Glib
```

These define the object-oriented figures which will represent places/transitions (*LoadFig*) and also the arcs (*BezFig*) in the graph.

```
import FigureCast
import BezFig
import LoadFig
```

The drawing view compatible with Gtk.

```
import GtkDView
```

The transformation computation functions are imported here.

```
import GraT
```

The necessarily tools for our editor.

```
import Tool
import SelectionTool
import GrabTool
import BezTool
import CreateTool
```

Miscellaneous functions to increase productivity, such as making buttons for tools and transformations.

```
import GtkUtil
```

Lastly, the definitions of our Petri net graph pieces.

```
import Petri
```

The Gtk library requires that we initialize the GUI.

```
main :: IO ()
main = do
  initGUI
```

For ease in building the components and layout of the GUI, we use the Glade [GNO98] library, bound to Haskell through the Gtk2Hs library.

Glade stores the GUI in an XML file which we must first load. Assuming the loading succeeds we can then proceed to get widgets from the file.

We are able to pull out any widget we want, by name, that is defined in the Glade file. We do this below, getting the important boxes, windows, drawing areas, and menu items so we may assign them functionality within our code.

```
mGlade ← xmlNew "gltest.xml"
gladeDescr ← case mGlade of
  Nothing → fail "Glade GUI file not found"
  Just x  → return x
let getWidget :: WidgetClass w ⇒
                 (GObject → w) → String → IO w
    getWidget = xmlGetWidget gladeDescr
w          ← getWidget castToWindow      "window"
drawArea   ← getWidget castToDrawingArea "drawingarea"
drawBox    ← getWidget castToHBox        "hbox1"
toolBox    ← getWidget castToVButtonBox  "buttons"
saveItem   ← getWidget castToMenuItem    "menuItemSave"
openItem   ← getWidget castToMenuItem    "menuItemOpen"
pdfItem    ← getWidget castToMenuItem    "menuItemExportPDF"
psItem     ← getWidget castToMenuItem    "menuItemExportPS"
makeWidgetScrollable drawBox drawArea
editState ← setupEditorState drawArea
onKeyPress w ∘ deleteHandle drawArea ∘ snd $ editState
onActivateLeaf pdfItem (exportHandler PDF w editState)
onActivateLeaf psItem  (exportHandler PS w editState)
```

The drawing knows how to save itself, without any further information. This is because the figures within the drawing are sufficient for the drawing to be saved.

However, the loading of a drawing is different. Loading requires the application to know what kind of figures may be in the drawing. Since the figures themselves define their parsing through their type, we only have to submit a list of figure types, indirectly passed in through/via undefined *error* values. We see the tail-polymorphic records (indicted by the "P" in their type-names) cited here with () in the spot reserved for tails.

```
onActivateLeaf saveItem $ saveHandler w editState
onActivateLeaf openItem $ openHandler w editState
  [toFigure (error "loadfig petri" :: LabelledFigP Petri s r ())
  ,toFigure (toConnFig (error "bezFig" :: BezFigP s r ()))
  ]
```

Next, we create a few "prototypes" of the figures we will want to appear on the canvas. Since this is a Petri net editor, we will need to create *LoadFig*s that represent the two types of nodes: places and transitions.

```
let placeProto = const $ fmap LabelFig $ mfix $ loadfig (Place 1) ()
    transProto = const $ fmap LabelFig $ mfix $ loadfig Trans ()
    toolButton' = toolButton toolBox editState
```

The selection and grabbing tools are conveniently already provided by the framework, as they are able to work on any kind of graph.

```
toolButton' "Select"    =<< (mkTool $ seltool ())
toolButton' "Grab"      =<< (mkTool $ grabtool ())
```

Additionally, we must also make tools and buttons to activate the tools based on the prototypes we have made.

```
toolButton' "Place"     =<< (mkTool $ createTool placeProto ())
toolButton' "Transition" =<< (mkTool $ createTool transProto ())
```

Here, a bezier figure is created, then updated using an object transformer: *compatibleConnFig petriCompatible*. This transformed bezier figure is then given to the bezier tool, which will handle the interactive creation of the figure. The object transformation enables us to enforce the bi-partite quality of the underlying graph.

```
toolButton' "Arc"       =<< (mkTool $
  beztool (fmap (compatibleConnFig petriCompatible) ∘ bezfig ()) ())
```

Also, we can construct buttons out of transitions as well. In this case, we reproduce the transition that is automatically invoked by selecting a transition through the GUI. However, in this case, since a transition is not interactively selected, for firing, one is selected using the *trans* selection computation.

```
mkTransButton "Fire Any"
  (selToGra trans >>= tokenTrans) editState toolBox
```

Lastly, we show the main window, and attach a handler so the application quits when the the window is closed. The *mainGUI* function starts the Gtk interface.

```
widgetShowAll w
onDestroy w mainQuit
mainGUI
return ()
```

7.3 Petri Net Discussion

In the previous sections, we have seen that the creation of the Petri net editor can be presented in a largely functional style. Even though the underlying graphical representation is built from an object-oriented system, the use of the *FigureLabel* and *LabelledFig* allows this freedom. Also, the *SelectM* and *GraT* selection and transformation monads allow expressive definitions of the firing of Petri net transitions.

To demonstrate the resulting system, we show two screen shots in Figure 7.1; the transition between the two states depicted can be achieved either by clicking the "Fire Any" button, or by directly double-clicking the firing transition. This tool allows free intermingling of transition firing between editing and layout steps; the Petri net shown has been assembled interactively in an entirely intuitive fashion.

Chapter 8

Future Work and Conclusion

8.1 Future Work

As an extension to the existing body of work, there are several areas where improvements and further research may be done.

Of primary importance is to develop a functional framework for canvas design, rather than borrowing existing ideas from the object-oriented body of work. There is some hope here using the functional reactive programming technique, which closely models Kleisli arrows. Such a technique has been used already in a graphical arcade game [CNP03]. However, this would be a large body of work, likely the scope of another Masters thesis itself.

Also, there may be some interesting possibilities extending the single pushout approach analogy we use for computations to the double pushout approach. This may involve a different arrangement of computation environments and ideas to compensate for the sharing of the matching morphsim.

The canvas framework also does not give equal treatment to the surrounding GUI, both in terms of functional style and integration. Since the GUI is mostly left up to the implementor to design, it may be beneficial to abstract the common GUI tasks out, and provide a coherent programming interface. The current GUI toolkit that is used is not completely functional, so this may also be one area where more functional design may be incorporated, once a functional GUI toolkit is available. This change would suggest also that more separation between the GUI toolkit and the framework is also needed. The ability to swap Gtk, Tk, or wxWidgets toolkits would be very useful and would increase general portability.

8.2 Conclusion

The framework that has been presented has been shown to provide a concise and extensible environment for the creation of graph editors. This has been shown directly in Chapter 7, where only a few pages of fully literate Haskell code are needed to create a completely functional editor and interactive animator for Petri nets with publication-quality drawings and intuitive interaction.

The technique which allows this to occur is the bridge from the object-oriented canvas framework to a style more suitable to functional programming. Such a bridge puts the succinct nature of functional programming to work for the developer.

Additionally, an EDSL for graph transformation was developed. The EDSL offers a small selection of combinators which can give rise to increasingly complex and expressive transformations. This method of building transformations is very powerful, and can actually replicate more well-known ideas in graph transformation theory, such as the single-pushout approach.

Bibliography

[BCM05] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Relating SPO and DPO graph rewriting with Petri nets having read, inhibitor and reset arcs. *Electr. Notes Theor. Comput. Sci.*, 127(2):5–28, 2005.

[Cai08] Cairo Developers. Cairo Graphics. `http://www.cairographics.org`, 2008.

[CE01] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *In Proceedings of the 2001 Haskell Workshop*, pages 41–69, 2001.

[CH93] M. Carlsson and T. Hallgren. *Fudgets - Graphical User Interfaces and I/O in Lazy Functional Languages.* Licentiate thesis, Chalmers University of Technology, Gteborg, Sweden, May 1993.

[Cha02] Manuel Chakravarty, editor. *Proc. Haskell Workshop 2002, Pittsburgh.* ACM Press, 2002.

[CNP03] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[EG98] Thomas Eggenschwiler and Erich Gamma. JHotDraw Implementation. `http://www.jhotdraw.org`, 1998.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[EHK+97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation, part II: Single pushout approach and comparison with double pushout approach. In Rozenberg [Roz97], chapter 4, pages 247–312.

[EL02] Levent Erkök and John Launchbury. A recursive do for Haskell. In Chakravarty [Cha02], pages 29–37.

[Ell07] Conal Elliott. Tangible functional programming. In *International Conference on Functional Programming*, 2007.

[Erw01] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(05):467–492, 2001.

[FGT92] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. Little theories. In *Automated Deduction, CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.

[GNO98] GNOME Developers. Glade Interface Builder. `http://glade.gnome.org`, 1998.

[Hud00] Paul Hudak. *The Haskell School of Expression, Learning Functional Programming through Multimedia.* CUP, 2000. ISBN 0-521-64408-9 (paperback), ISBN 0-521-64338-4 (hardback).

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 63–76, New York, NY, USA, 1992. ACM.

[Jon95] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 97–136. Springer, 1995.

[Kah01] Wolfram Kahl. A relation-algebraic approach to graph structure transformation, 2001. Habil. Thesis, Fakultät für Informatik, Univ. der Bundeswehr München, Techn. Bericht 2002-03.

[KL05] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. Draft; Submitted for journal publication; online since 30 Sep. 2004; Full version released 10 September 2005 at `http://homepages.cwi.nl/~ralf/OOHaskell/`, 2005. (last accessed 19 Oct. 2008).

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[KSFS05] Oleg Kiselyov, Chung-chie Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers. In *ICFP 2005, Intl. Conf. on Functional Programming*, volume 40(9) of *ACM Sigplan Notices*, pages 192–203. ACM, September 2005.

[LH96] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *ESOP '96*, volume 1058 of *LNCS*, pages 219–234. Springer, 1996.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *22nd POPL*. acm press, 1995.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[LO94] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.

[ML71] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[MV95] M. Minas and G. Viehstaedt. Diagen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *Proc. VL '95*, pages 203–210, 1995.

[Nor02] Johan Nordlander. Polymorphic subtyping in O'Haskell. *Science of Computer Programming*, 43(2-3):93–127, 2002.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, August 1991.

[PJ+03] Simon Peyton Jones et al. *The Revised Haskell 98 Report*. 2003. Also on `http://haskell.org/`.

[PJL92] Simon L. Peyton Jones and D. Lester. *Implementing Functional Languages: A Tutorial*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1992.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, Singapore, 1997.

[Sag99] Meurig Sage. FranTk – a declarative GUI system for Haskell, 1999. URL: `http://haskell.cs.yale.edu/FranTk/`.

[SPJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In Chakravarty [Cha02], pages 1–16.

[Tae99] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In *AGTIVE '99*, volume 1799 of *Lecture Notes in Computer Science*, pages 481–488. Springer, 1999.