UTRECHT UNIVERSITY

Analysing Pareci

Tim Soethout
tim.soethout@phil.uu.nl

August 30, 2013

Master's thesis
Computing Science
Department of Information and Computing Sciences
Utrecht University
Thesis number: ICA-3117901

Supervisor:
dr. Jurriaan Hage
Second Supervisor:
dr. Wishnu Prasetya

Universiteit Utrecht

Intership Supervisors:
ir. Eljakim Schrijvers
dr. Richard Forster

ELJAKIM
Information Technology bv

Abstract

Bugs in Pareci programs are often only found when the application is run. Another issue is that it is also easy to have many database queries, without the developer being fully aware of them. Using static data flow analysis we can find out many of these issues before we run the program and thereby reducing runtime errors and superfluous database queries.

This thesis describes what Pareci is, what its computational power is and how to parse it into a form to do program flow analysis on. We define how to create a Monotone Framework instance for Pareci, which is used to run analyses using a maximal fixed point worklist algorithm. We define three analyses that help respectively identify unresolvable bindings, superfluous database queries and incoherent variable type usage.

Acknowledgements

Apart from the efforts of myself, the success of any project depends largely on the encouragement and guidelines of many others. I take this opportunity to express my gratitude to the people who have been indispensable in the successful completion of this project.

Special thanks goes out to Saskia Ubbink, for all her support and encouragement.

Furthermore I would like to thank Eljakim Schrijvers and Richard Forster for letting me run free in their business and Pareci. Also I would like to thank my colleagues at Eljakim IT for the pleasant time there and the great, although sometimes odd, lunch conversations.

My supervisor Jurriaan Hage also deserves my appreciation for guiding me into the right direction by supplying an interesting research background and suggestions on how to approach the problems encountered.

Furthermore I would also like to thank my parents for their endless love and support.

Finally, I have greatly benefited from the skills and expertise of Jeroen Bransen and Alex Sijm.

Eljakim IT and Pareci

This thesis is a result of an internship at Eljakim Information Technology. The director Eljakim Schrijvers is my supervisor at the company. Eljakim IT is a medium-sized IT company based in Utrecht. They create all kinds of web applications for different kind of companies, but mainly non-profit and government. They are very interested in taking up all kinds of IT-related projects as long as interesting and doable.

Eljakim also has a lot to do with the Olympiad of Informatics, both nationally and internationally. During my internship I also had the opportunity to give a training to the team that represented the Netherlands during the International Olympiad for Informatics in 2013.

My second supervisor during the internship is Richard Forster. He is the boss of the web programming framework Pareci, which I did my research for. At the time of writing he is the president of the International Olympiad for Informatics and also working hard to get his product Pareci ready to be released onto the market.

Contents

1. Introduction

Eljakim IT (co-)develops a web development framework called Pareci that is used to create web applications. Pareci can be considered a programming language in itself as will become clear in chapter 2.

The main focus of this research will be to find ways to make Pareci more developer-friendly using proven research techniques as also used in modern compilers.

Due to Pareci's interpreted nature it suffers from the same problems as other interpreted languages. It has no strict types and exceptions only occur at runtime. This means that errors are only found when running the program and at worst only when the specific error condition occurs.

Many of these exceptions can be discovered by doing program analysis on Pareci programs. These analyses can be built into Pareci as dynamical analyses, but this will increase the overhead every time the program is executed. Therefore we will look into ways to make Pareci less error-prone by doing offline, static analysis.

To be able to analyse Pareci we need to be able to handle Pareci code. In order to do this a grammar, parser and Monotone Framework instance will be created as described in the next chapters.

1.1. Open Problems

This section discusses some known problem with Pareci, which can be (partially) solved using program analysis.

1.1.1. Runtime exceptions

Due to the interpreted nature of Pareci applications and the lack of type checking, all errors show up as exceptions at runtime. This means that to be sure that no exceptions are thrown during use, all possible uses of an application need to be tested. Of course this is not feasible due to the possibility of infinite program executions.

A simple example of many occurring runtime exceptions are referencing objects or fields on objects that do not exist. Since Pareci only tries to lookup these kind of references when they are accessed, these unresolvable references can lay low for quite a while until a user encounters them.

Many of these problems can be found beforehand by doing static analysis, for example in the form of type checking. For instance, references to identifiers that do not exist in the program can be reported before the program is actually used by a user.

1.1.2. Superfluous database queries

If a Pareci page contains a reference to a field that is not by default fetched from the database, the data layer does an extra database query for every additional field requested. This is by design to avoid unnecessary data transfer. This can be adjusted manually by inserting an *includeFields* parameter, which specifies the additional (foreign) fields to be retrieved. However, the developer can only find out about superfluous queries by knowing the system thoroughly or by investigating database logs.

A simple detection mechanism can be to scan the current page for requested fields and see if they are requested by default. This will be sufficient for most situations. However, it is possible that the resource is never used on the page due to dead code or that foreign fields are only used in specific cases. To handle these scenarios full a data flow analysis is required.

1.2. Approach

For parsing Pareci a formal syntactic form needs to be specified. The syntax is XML-based so in this sense the structure is straightforward. This means we can piggy-back on the XML specification [xml08] and use readily available parsers for this.

We do have to focus on the grammar of specific language parts falling outside the XML specification. How the XML and specific parts are handled can be found in chapter 3.

Another important part of parsing will be the parsing of the data model (in Pareci, the `models.yml`). With this information the analysis can check for presence of the correct fields and their types and is therefore more precise. For this we can reuse existing YAML parsers such as [yam13].

A nice detour is to also look at the computational power of Pareci in section 2.2. If we know what the expressiveness of the language is, then we know how difficult the analysis of Pareci is.

Statical analysis is discussed in the form of data flow analysis (see chapter 4). We discuss a notion of Monotone Frameworks and an instance [NNH04] for Pareci (in chapter 5). In section 5.2 three analyses relevant for Pareci are introduced, which solve the problems previously presented.

1.3. Bumps on the road

Pareci is an at the moment unreleased programming framework that is mainly used in-house at Eljakim IT. This means that there is a small user base and that lines to the developers are short, resulting in quick feedback. Pareci is in active development, therefore new versions and features are constantly released.

The changing environment, limited documentation and new language made it hard to quickly start with the actual analyses. Pareci is a complex language, unlike many

of the well-known programming languages, therefore it was hard to grasp the whole semantics quickly and correctly. Much time was taken by learning the ins and outs out Pareci and considering all Pareci possibilities that had to be taken into account for the analyses. There was also a strong learning curve in grasping the necessary static program analysis techniques, which took up much of the research time.

While working on the thesis I encountered a couple of bugs and usability quirks in Pareci, which were readily noted and sometimes immediately fixed. Due to the esoteric nature of my attempts and usage of Pareci I ran into situations that most users would not encounter. Also some bugs in documentation and implementation of the Utrecht University Attribute Grammar Compiler [DS05] were found and reported.

Due to time constraints only the approach of analyses on Pareci are reported in chapter 5. An implementation of these methods is also available, although not fully supporting all the analyses as described.

The following tools are implemented:

- a Pareci parser
- a binding and expression syntax parser
- an object model representation and parser
- a stand-alone monotone framework and worklist algorithm implementation supporting interprocedural analysis
- a monotone framework instance for Pareci
  - representing the complex program execution flow of a Pareci program
  - including the correct handling of scope, context, bindings and expressions
  - incorporating knowledge from the object model to improve the analysis results
- a liveness and used fields analysis

The source code is property of Eljakim IT and available to the graders.

2. Pareci

We will discuss the syntax and language features of Pareci in this chapter. Since Pareci is not a well known language at the time of writing, we will try to cover the most-used features to give the reader an intuition about Pareci. We will see Pareci pages, context and scope in section 2.1.

To determine which analyses techniques are required and strong enough to analyse a Pareci program, we will also look at the computational power of the language in section 2.2.

We end with a description of how to interpret a Pareci program, which contains the data structures and types needed to keep track of the program state and execution in section 2.3.

2.1. Language

Pareci is a declarative programming language, tailored for creating web applications. The back-end and the interpretation are written in PHP. This section will explain some basic details about the structure and semantics of Pareci, but is by no means an exhaustive definition.

The Pareci framework outputs HTML and JavaScript that is used on a client web browser. It has an AJAX based approach in communicating events to the web server. These can trigger actions such as updating database entries or refreshing (parts of) the page. Furthermore it connects to a database mainly through an extended version of the Doctrine Object Relational Mapper. Other database wrappers are also available for Pareci but not yet stable.

Throughout this thesis the following markup will be used for different Pareci aspects: `Widgets`, `properties` on widgets, `literals` and `bindings`.

2.1.1. Similar languages

Since Pareci is a dynamically typed language it shares properties with other dynamically typed languages. For many of these languages such as Perl [Jac05], PHP, Python [RHP05] and JavaScript [JMT09] soft typing work has been done. Many of the concepts discussed in these papers will also be applicable to Pareci.

For JavaScript [Thi05, JMT09], Python [Fri11] and PHP [CHH09] work is done in formulating type systems using Monotone Frameworks. The ideas used in typing these dynamically typed languages will prove particularly useful for creating a type system for Pareci.

2.1.2. Pages

One of the most important building blocks in Pareci is a Page. A Pareci application consists of pages which can be rendered as a web page or as a section inside another page. Pages are specified in an XML-format. Although the format is valid XML, some attributes contain "magic" strings that are parsed by the framework. An example is the binding and expression syntax, which we will see shortly.

Below, we see a simple example page `examplePage.xml` which renders into a web page with a simple link and a section placeholder. We will go through it step by step.

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action" >
  <Page.message>
    <Var id="varId" name="n" value="0" />
  </Page.message>
  <Page.resources>
    <Var id="newSectionName" value="='newSection' . {n}" />
  </Page.resources>
  <Page.actions>
    <a:Goto id="goto" target="{#newSectionName.value}"
            page="examplePage">
      <Var name="n" value="={n}+1" />
    </a:Goto>
    <Action_Log message="Goto completed." />
  </Page.actions>
  <Page.content>
    <Stack>
      <LinkButton text="='Load section ' . ({n}+1)"
                  onclick="{#goto}" />
      <Section name="{#newSectionName.value}" id="sectionid" />
    </Stack>
  </Page.content>
</Page>
```

Each XML-element corresponds to a widget defined in the framework, or a property of a widget. Elements containing a '.' (dot) corresponds to a property of a widget, i.e. $A.B$ is a property B of widget A. This construction also allows for properties to be assigned to widgets next to expressions or bindings. The attributes of the elements are matched to properties of the widget.

All properties can be set by using simple text, bindings or expressions. Expressions can be either a normal string or a more complicated expression, using arithmetic, boolean operators or references using bindings. The syntax and grammar of these expressions can be found in section 3.3. Bindings are references to objects on the page or fields of those objects. They are relative to the scope or the context which they are used in.

The first element of the example page is a `Page` widget. It also contains the namespace declarations for different kinds of widgets. In this case the page contains widgets and actions. The XML namespaces are not required, but make writing down the widgets more concise. Without the default namespace present it is possible to write out the full widget names such as `Widget_Page` and `Widget_Action_Goto`. In the example the notation `Action_Log` is used for a log action widget using the default namespace, but not using the a namespace.

The `Page` contains four much used properties (in element syntax) each with a different purpose: `message`, `resources`, `actions`, `content`. We will go through them one by one below.

The overall flow for this example page is that the user can click the `LinkButton` and then the placeholder `Section` gets filled in with another instance of the "examplePage" page, but with an increased value for n.

Messages

The `message` property contains possible arguments available to the page. They can have a default value, or can be passed in by a call from another page. In this example case the `message` contains a `Var` widget with `id n` and default `value 0`. The `Var`'s value is available in the current page context with its `name n`. The `Var` itself is available by its `id` in the page scope as `#varId`.

Resources

The `resources` property contains the local resources available to this page. Database objects can be placed in `resources`, including the possibility to do filters and specialized queries on the database.

The most frequently used resource is the `ObjectResource` widget. This widget binds to a PHP object via the property `object`, which is usually a wrapper around a database object. The `value` of the `ObjectResource` is then set to the result of the specified `method` of the `object` specified. For example:

```
<ObjectResource id="or" object="Person" method="getNew" />
```

This `ObjectResource` then has a `value` property of an empty Person object as specified in the associated object model. Two other default methods are `search` and `getSearchQuery` which allow for filtering the Person table in the object model. Both return a collection of the specified `object`. Note that in principle all PHP objects can be made accessible to the Pareci world.

Actions

The `actions` property contains all actions that are defined on the page.

The example page has a Goto action widget that fills the Section with id target with the specified page. The new page message is set to the child Vars of the Goto with corresponding name.

Actions can also be nested, for example with an ActionList element.

```
<a:ActionList id="saveAll">
   <a:Repeater items="{selected}">
     <a:Method object="{}" method="save" />
   </a:Repeater>
   <a:Goto page="selectPerson" />
   <a:Execute command="alert('Success.');" />
</a:ActionList>
```

Actions in an ActionList are executed in a sequential manner and can alter the state of the execution. This ActionList saves all items stored in the object selected, changes the page and displays an alert via javascript. {selected} is binding syntax and a reference some object in the current scope with identifier selected. Repeater acts as a foreach and iterates over all items contained in the selected binding. The child elements of the Repeater are the body of the loop and in this case performs the save method on each element[1]. The value of method corresponds to a function on the PHP object corresponding with the object property.

Essentially actions allow you to make calls to other actions and objects that alter the program state. Non-action widgets such as used in the page content, only exist in display data and can only alter the program state though calling actions.

Content

The content property is the actual rendered page content. Some widgets, such as the Stack widget, can contain more than one widget directly as children without the use of a property in element syntax. In this way widgets can be nested and multiple widgets can be displayed on the page. The Stack in the example contains a LinkButton and Section widget. The LinkButton in the example renders as a hyperlink that triggers the action with id equal to goto when clicked. Section is a placeholder here since its page property is not set; its page property can for example be set using an action, which will result in displaying the corresponding page at the location of the Section.

2.1.3. Context and Scope

Every binding in Pareci is relative to the context or to the scope. In most cases the scope and context of a widget are inherited from the parent in the page tree, sometimes changed or updated by a context set on the widget or a context changing widget type.

[1] {} is binding syntax for the current context, which is in this case one of the children.

In the case of an action widget the scope and context is always inherited from the calling widget. Basically before executing the widget the parent is set to the calling widget to make sure any scope and context lookups get resolved correctly.

There are three things of the parent widget that are relevant for the child and important for correctly resolving bindings: the parent context, the parent scope and the parent itself. All three of these must be provided and updated by the runtime.

Scope consists of all the ID's and `Var names` specified. Context is relative to the scope and used to set the context in which bindings are resolved for a block of widgets. It is also used to set the binding context for triggered actions.

Scope

Scope is defined by some widgets, such as `Page`, `Repeater`, `Paginator`, `Table` and `Tree`. As a rule of thumb, `Page` creates a scope, and widgets that iterate over items create a scope.

All ID lookups (such as `#id`) are relative to their scope. If an ID can not be found in the current scope the parent scope is searched until the top level scope such as a `Page` is reached. This is similar to variable lookups in many programming languages. Scope lookup only goes up to parent scopes and never down to children scopes.

In the case of widgets which iterate over their items, for each of the children a separate scope is created. This means that ID lookups can only be done to current and parent scopes, but never to scopes of siblings.

The scope can thus be represented by a stack containing for each level a set of the available ID's. The top item of the stack contains the ID's of the current level and the next item the ID's on the parent level and so on. An ID lookup will check the stack top-down until the requested ID is found. If a new scope is entered a new level of ID's is added to the stack. If a page scope is entered a new empty stack is used. This means that scope lookups do not go up further than the page and only ID's defined on the current page are found in the scope.

Context

Context in Pareci is the environment in which all the context relative bindings get resolved.

Every binding type can be reduced to a normalized (~) form as can be seen in table 2.1. The (~) corresponds to the current widget.

Table 2.1.: Reduction to normalized form

Binding syntax	**Normal form**
{~id}	~id
{id}	~context.id

... continued on next page

Table 2.1.: Reduction to normalized form (... continued)

Binding syntax	Normal form
{__name.id}	~_globals.name.id
{^id}	~_parent.id
{#id}	~_scope.id

As seen above with `{id}`, bindings without special characters #,~,^, are relative the current context. `{id}` is equivalent to `{~context.id}`, meaning that we are always looking up values. In this example it is the `id` value of the item in the current context.

Every widget has a context. It can either be set by its `context` property or inherited from its parent widget. Possibly the set `context` is relative to the inherited context of the parent. For the `Page` widget the context is always set to page `message` and therefore will contain the variable names declared there.

A special case for context inheritance are action widgets, where the context and scope are inherited from the calling widget. For example an action event that is registered by a `LinkButton` with an `onclick` event will use the context of the `LinkButton`. This can be seen as an implicit parameter passing the context when an action is executed. Also scope and parents are dynamic in this sense. `^id` binds to `id` property of the calling widget.

Widgets that iterate over items such as `Table` and `Repeater`, have a `context` of the respective item that is currently iterated over. The following fragment illustrates the use of such a widget where `listOfPersons` binds to an object collection with fields `firstname` and `surname`. The context of each of its children is one element of that collection.

```
<Table items="{listOfPersons}">
  <TextInput value="{firstname}" />
  <TextInput value="{surname}" />
</Table>
```

If we do not look at the dynamic context of actions and iterating widgets for a moment, we can rewrite every contextual binding to a more absolute binding relative to the scope. For example:

```
<Stack context="{#person.value}">
  <TextOutput value="{name}" />
</Stack>
```

is equivalent to:

```
<Stack>
  <TextOutput value="{#person.value.name}" />
</Stack>
```

If we do take dynamic contexts into account, then we need something more when looking at actions. Let us call the triggering of an action an $event$. A widget w can register an event e that can be triggered, which will result in the action a being executed. Upon triggering event e the context, scope and parent of w will be set to a, so that a can correctly resolve the bindings used in the execution of a as if it was a direct child of w (and thus inheriting its `context`). The crux here is that we need the context, scope and parent to correctly resolve the bindings used in the action. Since a single action can be triggered by multiple different widgets, we can not know beforehand what the calling context, scope or parent is and therefore we can not create an absolute binding beforehand. This is a dynamic feature of Pareci.

The `Trigger` action widget can trigger another action a and thus results in the execution of a which also needs to be in the correct context. This can be handled in the same way as any other action execution, taking into account that the context, scope and parent are also passed/set correctly.

2.1.4. Obscurities

Properties as Children

A property `p` on widget `W` can also have a child element in the XML tree with the widget name prefixed, as value. This way also widgets used as a property value.

```
<W p="value" />
```

≡

```
<W>
  <W.p>value</W.p>
</W>
```

For simplicity, let us call the specification of properties in the widget element tag the *attribute-wise specification* of properties and the above specification of properties in child elements the *element-wise specification* of properties.

Some widgets also support content directly as a child. The value is then bound to a property specified in the widget definition. This property is called the *logical child* and not the same for every widget; for example a direct child value for the `TextOutput` represents its `value`, while for `Stack` its children can only be `Widget`s and represent its `children`. Not all widgets have a specified logical child.

The code illustrates the equivalence of both ways of specifying properties.

```
<Stack>
  <TextOutput>
     text
  </TextOutput>
</Stack>
```

≡

```
<Stack>
  <TextOutput value="text" />
</Stack>
```

There is no corresponding attribute specification syntax for `children` of `Stack`, although it is possible to do something similar to: `children="text"`, `children="{#to}"` or `children="=({#w},{#w2})"` (Pareci list syntax). Although it is not fully supported this will result in the rendering of respectively a child cell with text content `text`, a clone of the widget with `id` equal to `to` and two clones of widgets with `id`s `w` and `w2`. The clones are identical instances of the referenced ones, also the context and scope of the cloned object are used.

Defining an action widget in the element specification of an event property is explicitly not allowed by the framework and results in an exception.

Using the above element and attribute syntax and the ability to have child elements as property value, properties can thus be one of the following number of types:

1. a simple text string, both in attribute and element specification
2. a binding, only attribute specification
3. an expression, only attribute specification
4. a widget, only element specification
5. a list of widgets, only element specification

Parent as context

The properties `context`, `condition`, `permissions` and `resources` are relative to the parent context. This means they should be evaluated as such instead of in the current widgets context. In other words,

```
<Stack context="{a}">
  <TextOutput context="{b}" condition="{c}" value="{d}">
    <TextOutput.resources>
       <Var name="v1" value="{e}" />
    </TextOutput.resources>
</Stack>
```

is equivalent to:

```
<Stack>
  <TextOutput context="{a.b}" condition="{a.c}" value="{a.b.d}">
    <TextOutput.resources>
       <Var name="v1" value="{a.e}" />
    </TextOutput.resources>
</Stack>
```

This makes sense because both `condition` and `permissions` determine whether a widget gets rendered or not. `resources` are available for use by the widget and therefore need to exist before the widget is rendered, and use the context of the parent of the widget in which it is contained. `context` is also required before other bindings can be resolved. All other properties such as `value` are resolved in the new context.

Assigning

It is not possible to assign a value to a field of an object that does not exist:

```
[..]
<ObjectResource id="person" object="Student" method="getNew" />
[..]
<a:Assign field="{#person.value.age}" value="=2" />
[..]
```

If class `Person` has no field `age` this will result in a runtime error when the `Assign` action is called, because the binding cannot be resolved.

It is possible to assign a new value to a `Var`:

```
[..]
<Page.message>
  <Var name="person" value="={#or.value}" />
</Page.message>
[..]
  <ObjectResource id="or" object="Student" method="getNew" />
[..]
  <a:Assign field="{person}" value="=2" />
[..]
```

Before the `Assign` action is called, the variable `person` refers to a `Person` object, but after the action is executed, it will become an integer value. Note the $=$ in the `value` of the `Var`, this expression makes sure that the `Var` can be reassigned. This is called a weak binding for Pareci. If the $=$ is omitted the variable resolves to the value of the `{#or.value}` and since the value of an `ObjectResource` is read-only it will result in an error.

Special Bindings

Globals Global values can also be queried by using the binding syntax, i.e. `{__application.name}`, which is equivalent to `{~_global.application.name}`, will return the application name. A couple of default available global stores exist, such as `__application`, `__authentication`, `__page`, `__datetime`, `__session`, `__cookies` and `__gds`. All but the last have specified available fields in the documentation.

__gds is the global data store in which the developer can store key-value pairs which are available throughout the whole application.

It is also possible that other global stores are registered and available in the application.

Getters and Setters Bindings are usually resolved by querying the associated PHP object that it is associated with. Consider binding {a.b}, which is equivalent to {~_context.a.b}. The symbol ~ is the current widget, thus ~_context.a refers to some a in the context of the current widget.

Different kinds of MemberAdapters are registered in a specified order to handle every part of the binding. {~_context.a.b} translates to PHP: widget->get_scope()-><..>(a)-><..>(b) given that there is a MemberAdapter that recognizes a and b on the object. For example if the object returned by <..>(a) is an array, <...>(b) will return the value associated with key b in that array.

Important to mention here is that the last registered adapter always tries to call getAbc() for binding abc. This also happens with get_scope() for _scope. So if you create a method on a PHP object which starts with get<X>, it is accessible in Pareci by x given the context of an instance of that object. Note the case difference, Pareci calls the php function with the first letter capitalized.

2.1.5. Multiple Pages

Embedding and changing pages

When a Pareci application is run, a single page is displayed. This page can contain more pages by using the Section widget. This widget contains a reference to another page in its page property. The page can be relative to the current page or absolute to the page directory (starting with a /)[2]. The file extension should be omitted. The page property can be set directly in the page definition or later by an action or a Goto action.

The Goto action changes the page element of a section. When its target is not specified this is the section in which it is contained, i.e. a page change. If the target is set to resolve to a Section as well, then the section will load the specified page.

Passing arguments to pages

Interesting here is the possibility to pass values to the newly loaded page. Variables made available in the Page's message can be set by placing corresponding Vars in a Section or Goto widget. For example:

```
<Section name="newSection" page="newPage">
```

[2] It is also possible to have more page directories, they will be used in the order specified in the Pareci settings.

```
  <Var name="text" value="This is a new page" />
  <Var name="backLink" value="backPageName" />
</Section>
```

The values of the above Var widgets are set on the specified newPage to the Var widgets in the page's message there with the same name. This is a copy, so changes on the new page are not reflected on the calling page. This way it is possible to pass arguments to a page which can do some computation and display the result, or pass it to another page.

The default values set on the Page's message are overwritten with the specified variable values only if the default value was set with a weak binding (i.e. ={a} instead of {a}).

Recap

We have seen the basic building blocks of a Pareci page. Using bindings we can refer to other parts of the page and (data) resources.

An important part and power of Pareci lies in the scope and context. Scope and context allow for easy reuse of code by using the same widget declarations for different instances with different values due to different scope or context. A simple example of this is a table which repeats its child widgets each time in the context of the next data object in its items. Another is the different context used when calling an action from a widget with a different parent context.

We have also looked into different ways to specify widget property values: using element and attribute syntax, and via logical children.

In the end we described how to reference other global specified variables and an extra way to refer to values on objects.

2.2. Computational power

This section discusses the computational power of Pareci. The possibility of recursion is a good indicator of the expressiveness and complexity of a programming language. We will first look at the possibility to do simple recursion in Pareci and then at what this means for Pareci.

2.2.1. Recursion

Recursion with Actions

We first approach recursion by using actions and a single page. An ActionList is used to simulate function calls. This simple page tries to count down from n to 0 by doing a recursive call. The computation is started by clicking a button with the text Do Recursion

The computation corresponds roughly to the following Haskell code:

```
decrease :: Int -> Int
decrease 0 = 0
decrease n = decrease (n-1)
```

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action"
      cacheBindings="false">
  <Page.resources>
    <Var id="n" value="{#input.value}" />
  </Page.resources>

  <Page.actions>
    <a:ActionList id="method">
      <a:ActionList id="recCall" condition="={#n.value}>0" >
        <a:Assign field="{#n.value}" value="={#n.value}-1" />
        <a:Trigger id="rec" target="method" />
      </a:ActionList>
      <a:Refresh target="{#output}" />
    </a:ActionList>
  </Page.actions>

  <Page.content>
    <Stack >
      <Debug id="output" value="{#n.value}" />
      <NumberInput id="input" label="Input" value="=10" />
      <LinkButton onclick="{#recCall}"
                  text="Do recursion" />
    </Stack>
  </Page.content>
</Page>
```

This approach runs into problems when trying to use the page. There are two issues with the framework.

- Calling trigger `target` by binding `{#method}`, results in infinite unfolding of actions.

- Calling by name `method` makes the page work, but when the action is triggered by the button press Pareci crashes the second time the action is called. This is related to the dynamic parent getting set or unset incorrectly. Basically the called action is its own parent and the interpreter loops when resolving the bindings.

If we follow the specification of Pareci this scenario should be supported, the implementation in fact does not support it at the moment of writing.

Recursion with Pages

We will now look at an approach that uses a page as a function and their messages as arguments to calculate a fibonacci sequence. No actions are involved.

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action">

  <Page.message>
    <Var name="n" value="=90" />
    <Var name="first" value="=0" />
    <Var name="second" value="=1" />
  </Page.message>

  <Page.content>
    <Stack layoutMode="Flat">
      <NumberOutput value="={first}+{second}" />
      <Section page="fibonacciPages" condition="={n} != 0" >
        <Var name="n" value="={n} - 1" />
        <Var name="second" value="={first}+{second}" />
        <Var name="first" value="={second}" />
      </Section>
    </Stack>
  </Page.content>
</Page>
```

When this page is displayed it will render the first ninety numbers of the fibonacci sequence (minus 0). The n variable is used as an accumulator to make sure the computation terminates.

One drawback from this approach is that you can not use function results in computations. Results are only printed to the page. If the PHP debugger XDebug is active then the default PHP function nesting level is reached when setting n higher than 8. If the limit is increased or the debugger disabled Pareci happily computes higher numbers.

Recursion with Goto

This approach passes the result to the current page by using the `Goto` action instead of nesting `Section`s.

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action"
      onloadServer="{#onload}">
  <Page.message>
    <Var name="n" id="value" value="=90" />
    <Var name="first" value="=0" />
    <Var name="second" value="=1" />
  </Page.message>
```

```
  <Page.actions>
    <a:ActionList id="onload">
      <a:Goto page="fibonacciPagesGoto" condition="={n} > 0">
        <Var name="n" value="={n} - 1" />
        <Var name="second" value="={first}+{second}" />
        <Var name="first" value="={second}" />
      </a:Goto>
    </a:ActionList>
  </Page.actions>

  <Page.content>
    <NumberOutput value="={first}+{second}" />
  </Page.content>
</Page>
```

In this approach the page replaces itself with a new instantiation of itself and thereby setting its two fibonacci counters to appropriate new values. This variant uses the `onloadServer` property of the page to trigger the computation. The n variable denotes that is will calculate the $n+2$th fibonacci number. Again the PHP function nesting level is reached for any n larger than 4, but it works fine if this restriction is lifted.

If the `onloadClient` is used instead, the client will trigger the action call, this results in seeing each page being displayed and replaced until the $n+2$th fibonacci number is reached. Here the nesting limit is not reached, since every next call results in a new request with new values, which is handled by a fresh server process.

Recursion with Goto and return values

A naïve exponential time fibonacci algorithm corresponding with the following Haskell code is implemented in the Pareci page (`fibonacciPages2.xml`).

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

The corresponding Pareci page is:

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action"
      onloadServer="{#function}" >

  <Page.message>
    <Var name="n" value="=5" />
    <Var name="result" />
    <Var name="result1" />
    <Var name="result2" />
```

```
    <Var name="putResultName" value="result1" />

    <Var name="parentSectionName"
         value="{#section.value.logicalParent.scopePath.0}" />
  </Page.message>

  <Page.resources>
    <ObjectResource id="section" object="Widget_Section"
                    method="getSection">
      <Param name="sectionName" value="{~scopePath.0}" />
    </ObjectResource>
  </Page.resources>

  <Page.actions>
    <a:ActionList id="function">

      <a:ActionList condition="{result}">
        <!-- Return value to parent -->
        <a:Log message="='return branch='.{#section.value.name}" />
        <a:Trigger target="return" section="{parentSectionName}">
          <Var name="{putResultName}" value="{result}" />
        </a:Trigger>
      </a:ActionList>

      <a:ActionList condition="=NOT {result}">
        <a:Log message="='functi branch='.{#section.value.name}" />
        <!-- Base cases -->
        <a:Goto condition="={n} == 0 OR {n} == 1"
                page="fibonacciPages2">
          <Var name="putResultName" value="{putResultName}" />
          <Var name="result" value="=1" />
        </a:Goto>

        <!-- Recursive case -->
        <a:ActionList condition="={n} > 1">
          <a:Goto target="={#section.value.name}.'_l'"
                  page="fibonacciPages2">
            <Var name="n" value="={n}-1" />
            <Var name="putResultName" value="result1" />
          </a:Goto>
          <a:Goto target="={#section.value.name}.'_r'"
                  page="fibonacciPages2" >
            <Var name="n" value="={n}-2" />
            <Var name="putResultName" value="result2" />
```

```
            </a:Goto>

            <a:Goto page="fibonacciPages2">
              <Var name="putResultName" value="{putResultName}" />
              <Var name="result" value="={result1}+{result2}" />
            </a:Goto>
          </a:ActionList>
        </a:ActionList>
      </a:ActionList>

      <a:ActionList id="return" public="true" />
    </Page.actions>

    <Page.content>
      <Stack>
        <Section name="={#section.value.name}.'_l'" id="leftBranch" />
        <Section name="={#section.value.name}.'_r'" id="rightBranch" />
      </Stack>
    </Page.content>
</Page>
```

This page has subsections creating the sub-results. If the result of the current page is calculated it replaces itself with a version that contains the result. With a `Trigger` action the result is passed to the parent.

The parent section `parentSectionName` is resolved by a doing a call to a static method of the `Section` PHP class to get the current section and then the $\mathrm{logicalParent}$.

To calculate a sub-result a `Section` is created with a corresponding result `Var`. If the base case is reached ($n \in \{0, 1\}$) the base value is set on the parent by using the public action by using the Pareci feature where the `Var` of `Page`'s `message` can be set by specifying a `Var` with the same name inside the `Trigger`. The child actions of an `ActionList`s are carried out sequentially and are therefore the results in the first branches get computed first and inserted in the current `Section` and therefore are available to the rest of the `ActionList`. `condition`s are used to make sure that either the sub-results are computed or that the result is passed up.

The computation is done depth first and the answer is ultimately displayed in the section where the page is first embedded.

The function can be "called" with:

```
<Section id="fiboMethodCall" name="fibo" page="fibonacciPages2"
         condition="=ISNULL {result}" >
  <Var name="n" value="=4" />
  <Var name="putResultName" value="result" />
</Section>
```

The calling `Section` should contain a variable in the page message with `name` equal to `result` and a public `ActionList` with id 'return'. The condition makes sure that

the computation is not done again once the result is known.

The way the variables are passed here is similar to the way an attribute grammar works. First the left branches are calculated as a whole (depth first), then the right. For example the following log information is output for the 4th fibonacci number:

```
functi branch=fibo
functi branch=fibo_l
functi branch=fibo_l_l
functi branch=fibo_l_l_l
return branch=fibo_l_l_l
functi branch=fibo_l_l_r
return branch=fibo_l_l_r
return branch=fibo_l_l
functi branch=fibo_l_r
return branch=fibo_l_r
return branch=fibo_l
functi branch=fibo_r
functi branch=fibo_r_l
return branch=fibo_r_l
functi branch=fibo_r_r
return branch=fibo_r_r
return branch=fibo_r
return branch=fibo
```

Pareci seems to be able to do this recursion for bigger fibonacci numbers without running out of memory, but is limited by the PHP timeout of default 60 seconds. Of course the algorithm is very inefficient and it quickly takes too much time to compute it. For example the 10th number takes about 50 seconds to compute on the testing machine and this increases exponentially with every increase of the number. Of course this has to do with the inefficiency of the algorithm used. There is no caching or sharing done and thus every branch calculates the same fibonacci numbers over again.

2.2.2. Turing Completeness

Given the nature of Pareci, the possibilities of actions and the possibility to do recursion as done in the previous sections, it seems likely that Pareci is in fact Turing Complete [Sip06] and that a proof for this can be formulated.

It is interesting to know what the computational power of Pareci is. When we know the power of Pareci, we can determine which analysis techniques are complex enough to model and analyse Pareci programs.

Turing Machines

In his 1936 article [Tur36] Alan Turing describes a machine, the Turing Machine, which is capable of computing everything that is computable. This machine is in essence a

very simple device, consisting of a ticker tape with data cells, a reading and writing head and a mechanism to move the head over the tape.

If we are able to devise a Turing machine inside Pareci, the whole computational power of the Turing Machine is also available to Pareci and therefore Pareci would be a Turing Complete language.

Of course Pareci is Turing Complete when we take into account the possibility to do calls to PHP functions, since PHP is Turing Complete. Below we will show that Pareci is also powerful enough to simulate a Turing Machine enough without excessive calls to PHP functions beside the use of an external data container.

Simulating a Turing Machine with Pareci

To correctly simulate a Turing Machine we must support all actions a Turing Machine supports. Let us first look at the ticker tape that is used for the data. We can see the ticker tape as three parts, one cell on which the head currently resides, one part to the left of it and one part to the right. We can simulate this as three Pareci `Var` widgets. The left and right parts of the tape are implemented by using a PHP object `ArrayList` which is used as a simple stack on which we can pop from and push out values. These are handled by using an `ObjectResource` to instantiate the `ArrayList` with the initial configuration of the ticker tape as seen below.

```
<Page.message>
  <Var name="tapeLeft" value="={#arrayL.value}" />
  <Var name="current" value="=0" />
  <Var name="tapeRight" value="={#arrayR.value}" />
</Page.message>
<Page.resources>
  <ObjectResource id="arrayL" object="Util" method="getNewArrayList" >
    <Param value="=(1,1,1)"/>
  </ObjectResource>
  <ObjectResource id="arrayR" object="Util" method="getNewArrayList">
    <Param value="=(1,1,1,0,0)"/>
  </ObjectResource>
</Page.resources>
```

In this case we have a ticker tape with values $\langle 1,1,1,0,1,1,1,0,0 \rangle$ and the machine's head on the first 0 in the sequence.

The moving of the head is handled by two actions, one for each direction:

```
<a:ActionList id="moveRight">
   <a:Method method="append" object="{tapeLeft}">
     <Param value="{current}" />
   </a:Method>
   <a:Method id="popRight" method="shift" object="{tapeRight}" />
   <a:Assign field="{current}" value="={#popRight.result}" />
```

```
</a:ActionList>
<a:ActionList id="moveLeft">
    <a:Method method="unshift" object="{tapeRight}">
      <Param value="{current}" />
    </a:Method>
    <a:Method id="popLeft" method="pop" object="{tapeLeft}" />
    <a:Assign field="{current}" value="{#popLeft.result}" />
</a:ActionList>
```

For a move of the head one cell to the right, the value under the head gets appended to the left part and the value on the top of the right stack gets used as the new current/head value. The move to the left is done symmetrically. Note that for the handling of the right tape part we make use of the `shift` and `unshift` functions which are analogous to relatively append and pop functions, but work on the other end of the `ArrayList`. Since we only use one pair of these functions per side, this can trivially be rewritten to make use of the same functions. For displaying purposes we use this approach.

Each Turing Machine also has a state the controller is in, this is denoted by another variable `state`. The starting state of our simulation is 1. The machine wil halt as soon as it reaches state 0.

```
<Var name="state" value="=1" />
```

The Turing machine also has a configuration which consists of a set of 5-tuple $\langle \mathrm{s}, \mathrm{r}, w, \mathrm{d}, \mathrm{n} \rangle$; the current state s, the symbol on the tape at the head r, the symbol to write w, the direction to travel d and the new state of the machine n. Given the current state s and the symbol read r from the tape the behavior of the machine $\{w, \mathrm{d}, \mathrm{n}\}$ can be determined by looking them up in the configuration.

We represent this in Pareci with a big set of actions with appropriate conditions on them, to control which rule is relevant. Below is an example of corresponding with a configuration with one rule $\langle 1, 0, 1, \mathrm{r}, 2 \rangle$

```
<a:ActionList id="run">
 <!-- Begin Machine configuration table -->
 <a:ActionList condition="=(NOT {done}) AND {state} == 1" >
   <a:ActionList condition="=(NOT {done}) AND {current} == 0" >
     <a:Assign field="{current}" value="=1" />
     <a:Trigger target="{#moveRight}" />
     <a:Assign field="{state}" value="=2" />
     <a:Assign field="{done}" value="true" />
   </a:ActionList>
 </a:ActionList>
 <!-- Possibly more rules here -->

 <a:ActionList id="nextStep" condition="={done}" >
```

```
    <a:Goto page="turingMachineMult">
      <Var name="tapeLeft" value="{tapeLeft}" />
      <Var name="current" value="{current}" />
      <Var name="tapeRight" value="{tapeRight}" />
      <Var name="state" value="{state}" />
    </a:Goto>
  </a:ActionList>
</a:ActionList>
```

The first `ActionList` with `id` `run` is used to execute the next step of the Turing Machine. In this case the second and third `ActionList`s only get entered if their conditions are met, corresponding to being in state 1 and reading a 0. A boolean variable `done` is added to make sure that only one rule is executed per step when multiple rules are present inside `run`. Also the tape variables get updated for a move of the head to the right by the triggering the action `moveRight`, the state gets updated to the new state 2.

After the rule for the current step is processed, the page replaces itself with a `Goto` widget with the relevant tape and state information passed. If no rule is applied (in the case the state is 0), then the `done` variable stays `false` and the whole system halts.

In the appendix two example programs can be found based on this principle. The first example found in appendix A.1 starts with the machine's head between two numbers encoded in unary notation, after halting, it will have one number on the tape which is the sum of the two. The second machine found in appendix A.2 has a larger program and calculates the product of two binary encoded numbers. Both run correctly.

We can see that Pareci can mimic a Turing Machine with constant overhead, thus Pareci is Turing Complete.

Turing Completeness means that we need a powerful analysis technique to be able to do correct and concise enough static analyses on Pareci. This will be achieved with Monotone Frameworks for doing data flow analysis on Pareci programs in section 4.2.

Next we discuss what a Pareci interpreter Pareci should support, to evaluate a Pareci program correctly.

2.3. Interpreting Pareci

To be able to interpret and evaluate a Pareci application correctly a couple of things are needed.

Let us define a Pareci application $\mathrm{A} = \langle \mathrm{P}, \mathrm{p}, \mathrm{G}, \mathrm{M} \rangle$ where

- $\mathrm{P} \in \mathcal{P}(\mathrm{PageName} \times \mathrm{TreeRepresentation})$ are the available pages
- $\mathrm{p} \in \mathrm{P}$ is the initial page
- $\mathrm{G} \in \mathcal{P}(\mathrm{StoreName} \times \mathcal{P}(\mathrm{KeyValuePair}))$ are the available global stores
- $\mathrm{M} \in \mathcal{P}(\mathrm{ObjectName} \times \mathrm{ObjectType})$ is the object model.

When a Pareci application is started the following events happen in order:

1. The initial page $\mathfrak{p}$ is loaded.
    1. All declared widgets loaded on the initial page are instantiated. At this moment all variables in the `Page` `message`, resources and local variables are available.
    2. For every widget w the scope is set by taking the scope of its direct parent possibly updated by its own scope if w is a scope creating widget.
    3. For every widget w the context is set by taking the context of its direct parent and possibly updating it with its own `context` if it is set.
2. Any `onloadServer` event is executed first.
3. The page is rendered on the client.
4. The `onloadClient` event is executed.
5. An action can be executed by a user triggering an event.
    1. An instantiation of the corresponding action with the correct context, scope and parent is executed.
6. Possibly another event can be triggered.

Triggered Action $\mathfrak{a}$:

1. If $\mathfrak{a}$ is a scoped widget, the scope is updated.
2. The `permissions` and `condition` are checked. If false then the action is not executed.
3. The `resource` is instantiated.
4. The `context` is set for all other bindings to use.
5. The `fire` method of the action is executed.
    1. Any bindings are looked up.

Binding lookups:
For binding lookups the following should be available:

- Binding $\mathfrak{b}$
- Object Model $\mathcal{M}$
- Global store $\mathcal{G}$
- Current widget w with associated context $\mathfrak{c}$, parent $\mathfrak{p}$ and scope $\mathfrak{s}$.

First we rewrite $\mathfrak{b}$ to its normal form $\mathfrak{b}$ as seen in table 2.2. The normal form $\mathfrak{b}$ can have a limited number of forms:

Table 2.2.: Resolving of Bindings

Form of b	**Resolved by**
`~a`	Get property `a` from widget w.
`~context.id`	Resolve `id` in the context c of w using M.
`~_globals.name.id`	Find value with key `id` in global store under `name` in G
`~_parent.id`	Resolve `~id` on parent p of w
`~_scope.id`	Find `id` in scope s

M and G can be determined before running the application and are immutable. The values of c, p and s can also be determined statically when non-action widgets are used, but since actions use the context of their caller we have to deal with dynamic c, p and s there.

Let us delve a bit into the $\mathrm{TreeRepresentation}$. This tree representation represents a Pareci page. The root of the tree is the page widget, for which a couple of properties in the form of widget XML-elements are set. We also need a set of allowed Widgets $\mathcal{W}$, which contains all of the actual implementation of available widgets.

2.4. Conclusion

We have seen many of the language constructs that are used to create Pareci programs. The computational complexity of Pareci is also determined to be Turing Complete and therefore we need powerful analysis techniques. We will look into analysing Pareci in the form of Monotone Frameworks in chapter 5. In the next chapter we will first look at how we can represent and read in Pareci programs.

3. Parsing Pareci

To be able to start analysing Pareci programs, we need to represent them. This chapter describes this. We first look at how to represent the structure of a Pareci program in custom data types in section 3.1, in this case Haskell data types. Then we look at how to parse the XML into these data types in a bidirectional sense using Picklers in section 3.2, then we will discuss how we can parse the expression and binding syntax in section 3.3.

3.1. Pareci data types

The representation of Pareci has gone through a couple of stages of varying complexity and correctness. We will describe the main variants here. The first variant is to create a data constructor directly corresponding with each Pareci widget. We will discuss two flavours for this first approach: One more restrictive and one more loose.

The second and final variant generalises over widgets and stores information about their types separately.

3.1.1. Constructor per Widget

Restrictive

We will construct a simple data type which has a constructor for each widget. Each constructor has fields associated to the widget's properties.

Consider this simple Pareci page fragment.

```
<Stack id="pageId">
  <TextOutput id="varId" />
</Stack>
```

A first attempt is to represent the different widgets by creating a parameterized generic algebraic data type.

```
data Stack
data TextOutput
data Widget a where
  TextOutput :: {vId :: String} -> Widget TextOutput
  Stack      :: {sId :: String, children :: [Widget a]} -> Widget Stack
```

The problem with this approach is that while a `Stack` can contain multiple children, all the children of `Stack` need to be of the same widget type. while in Pareci a `Stack` can contain more than one specific respresented widget type. This approach is thus too strict. Therefore, we need another approach, allowing for more mixed child widget types. It needs to allow more than Action Widgets inside `Page.actions`.

Less restrictive

Another less restrictive way is to represent the code fragment above as the following Haskell data type. Note that all widgets have some common base fields.

```
data Base       = Base       { id :: String }
data TextOutput = TextOutput { base :: Base }
data Stack      = Stack      { base :: Base
                             , children :: [TextOutput] }

fragment = Stack { base = Base { id = "pageId"}
                 , children =
                     [TextOutput { base = Base { id = "varId"} }]
                 }
```

This way only `TextOutput` widgets can be added as children. To be able to have multiple kinds of widgets, we introduce a constructor for each widget and let the children contain any widget.

```
data Widget = Base       { id :: String }
            | TextOutput { base :: Base }
            | Stack      { base :: Base, children :: [Widget] }
```

To be able to extend the supported widgets, we create this `Widget` data type using Template Haskell, a meta-programming tool that generates valid Haskell code (designed by Sheard and Jones [SJ02]). Since we use Template Haskell it is not necessary to model inheritance using a `Base` widget like above. From a structure that does contain the inheritance hierarchy, we can generate the widgets with all their inherited properties.

The generated Haskell data types have the following pattern.

```
data Widget = TextOutput { id :: String }
            | Stack      { id :: String , children :: [Widget] }
```

In this case we allow all widgets as children for `Stack`, while this is not allowed by Pareci. An action widget can not be a child of `Stack`. Dealing with the actual valid instances of the more restrictive data types is done in the actual analysis part.

Property Values

Instead of using `String`s to represent all property values we can make it corresponding more directly to how Pareci works. If we implement it as `Maybe String`, we can represent whether the value has an actual associated value in the XML or not. We use the `Bindable` type synonym for this:

```
type Bindable = Maybe String
```

Pareci experts will immediately see a problem here: property values can not be represented correctly by simple strings.

We can differentiate between a couple of scenarios as discussed in section 2.1.4, and therefore determine types of Widget property values.

Property values can be:

- not set, sometimes having a default value
- simple text, unresolved bindings or expressions
- widgets in the XML-tree

We can thus have either actual textual information in the form of text, bindings and expressions, or actual widgets. To be more precise we have a distinction between textual property values and actual resolved property values. Pareci internally stores the textual property values and replaces it with a reference to the resolved entity when requested.

When actually using this representation we create a distinction between property values that are directly input after the first parsing process and the property values used during analysis.

We define data type `UPropVal` to represent the first unparsed version of property values.

```
data UPropVal = NotSet
              | Unparsed String
              | UResolved [Widget UPropVal]
```

`NotSet` is used when the property has no value set in the XML. `Unparsed` wraps a `String` that is found corresponding with a property, this can either come from attribute syntax, containing a string, binding or expression or from simple text contained in element syntax. The last case, `UResolved` is used to insert children widgets that are directly found and parsed from the XML-tree.

For analysis we use a more specific type representing parsed property values:

```
data PPropVal = PV PropertyValue
              | Resolved (OneOrMany (Widget PPropVal))
```

Constructor PV wraps a parsed PropertyValue, which contains variants for not set values, bindings and expressions. This is a parsed version in which we will consider in more detail below when discussing Pareci bindings and expressions in section 3.3. Resolved denotes an already resolved widget, and makes a distinction between single widgets (One) or a list of widgets (Many). This distinction is made because some widget properties only allow single widgets. Ideally the parser uses this distinction to not loose information in the parsing step.

Given a binding and expressions parser we can trivially translate between these two property value types. These parsers are discussed below in section 3.3.

The alert reader might have noticed that we also need to update the definition of Widget to take into account two different kinds of widget value types. We now have a widget definition along the lines of:

```
data Widget a = TextOutput { id :: a }
              | Stack      { id :: a , children :: [Widget a] }
              [..]
```

3.1.2. Generalised approach

There are a couple of shortcomings with the previous approach, such as:

- Because the different widgets are hard coded in the Haskell data type, new kinds of widgets can not be added without recompiling and explicitely specifying how the analysis works for the new widgets.
- There are shortcomings with widget types, inheritance and specifying the types of properties.
- Picklers implementation as discussed in section 3.2 will turn out th be harder and have to be generated using Template Haskell accordingly. It is also harder to deal with element and attribute syntax, see section 3.2.

To solve this we use a more general approach to modelling widgets. We do not encode widgets directly in a data constructor but create a notion of what a widget is by defining widget definitions and a notion of widget instances.

```
data WidgetDefinition =
  WD
  { name :: WidgetName
  , parent :: Inherit
  , isAbstract :: Abstract
  , properties :: PropertiesDefinition
  , logicalChild :: Maybe PropName
  }
```

A widget definition consists of a widget name, a possible parent definition, a boolean flag indicating whether the widget is abstract, i.e. not directly available as a widget, the value properties in a map, containing the property name and type, and a logical child denoting which property the direct child elements of the widget's XML element should be interpreted as.

The widgets properties also contain type information, which we will discuss in more detail later on in section 5.2.1.

Analogous to the definition we have the widget instance:

```
data WidgetInstance a =
  WI { instanceOf :: WidgetDefinition
     , propertyValues :: PropertyValues a
     }
```

An instance contains the widget definition it is an instance of and has a map filled with property names and values. It is parameterized with a type variable `a` to be able to have instances for both `UPropVal` and `PPropVal`. We also make it an instance of `Functor` so that we can easily provide a functor to translate from `WidgetInstance UPropVal` to `WidgetInstance PPropVal` which we can use in the analysis.

For convenience we introduce the type synonyms `UnparsedWidget` and `ParsedWidget` for respectively `WidgetInstance UPropVal` and `WidgetInstance PPropVal`.

This definition is also closer to the way the widgets are implemented in Pareci than the constructor-per-widget approach in section 3.1.1. I have adapted the program *PareciSchemaGenerator*, already provided by the Pareci developers to create XML-schema files to recognise correct Pareci Pages, to also support outputting the widget definitions in valid `WidgetDefinition` Haskell code. These can be readily used by the actual implementation.

This approach is less precise than the other approach and does not make full use of the possibilites of representing widgets more strictly using the Haskell type system, but makes up for this by having widget definitions and instances available as values during runtime, making it possible to add new widgets without changing the whole program. Both the widget type and the corresponding XML-picklers (as discussed in section 3.2) become less complex and thereby no longer require Template Haskell to work. It also becomes easier to add type information to the the widget properties.

3.2. From and to XML

The next step is to get from the Pareci XML pages to the Haskell data types. There are a couple of approaches to do this in Haskell. One of the more mature packages is Haskell XML Toolbox [hxt12] (HXT) which uses the nice concept of Picklers. Originally this came from a Functional Pearl by Andrew Kennedy [Ken04].

A pickler is a bidirectional translator between two data forms, in this case between XML data and custom data types. It consists of two functions:

```
data PU t a = PU { unpickle :: t -> a
                 , pickle   :: a -> t
                 }
```

In our case we need to have a pickler of `PU XmlTree Widget` for some type `XmlTree` which makes it possible to translate the types back and forth.

The advantage a pickler gives us is that we can easily convert the Haskell representation of our widget back to a valid Pareci page. This might be useful in the future when implementing code optimisations.

Since the `XmlPickler` from HXT is created for parsing program generated XML and therefore always handles XML Elements in a fixed order, it is not suitable for parsing Pareci pages, since developers do not all use the same order for their elements in for example specifying the `Page.message`, `Page.actions`, `Page.resources` and `Page.content`. For this reason we chose to use the `xml-picklers` package, which does not have this restriction. It does this by trying to pick all child elements and consuming only those that succeed, leaving the remainder for consecutive picklers.

Another possibility is to write a pickler that can unpick every permutation of a list of given picklers. Baars, Löh and Swierstra have defined a way by using lazy evaluation in to create efficient permutation parsers [BLS04]. This could be a good starting point in creating permutation parsers in HXT, but we chose a different derived pickler package discussed below.

3.2.1. A picklers for constructor per widget data type

We first give an example for the widget per data constructor version from section 3.1.1 to illustrate how the library works. The `xml-picklers` package uses the following `PU` data type:

```
data PU t a = PU
   { unpickleTree :: t -> UnpickleResult t a
   , pickleTree   :: a -> t
   }
```

Here an `UnpickleResult` can either give a result and a remainder, no result , or an `UnpickleError`. Note that there is a remainder here. This means that unpickling does not necessarily need to consume the whole input.

To create specific picklers for parts of data we can use the `xpWrap` pickler combinator, which comes in handy when handling data constructors. The function `xpWrap`[3] takes two functions to translate back and forth to the projected type and uses it to transform a pickler of type `PU t a` to `PU t b`.

Consider:

```
xpVar :: PU t Widget
```

[3] `xpWrap :: (a -> b) -> (b -> a) -> PU t a -> PU t b`

```
xpVar = xpWrap (\ id      -> Var id)
               (\(Var id) -> id)
               ...
```

We use `xpWrap` to remove and add the `Var` constructor. This means we must provide a pickler of type `PU t String`. Since we want to create XML of the following form `<Var id="id" />`, we can use the `xpElem`[4] pickler combinator. This combinator takes an element name ("Var"), a pickler for the element attributes ("id") and a pickler for the element child elements (Nothing in this case).

To complete `xpVar` we use a specialised version `xpElemAttrs` of `xpElem`, which handles an element with only attributes an no child elements.

```
xpVar :: PU [Node] Widget
xpVar = xpWrap (\ id      -> Var id)
               (\(Var id) -> id)
               $ xpElemAttrs "Var" (xpAttr "id" xpString)
```

`xpAttr` handles the attribute whose content is handled by `xpString`. Note that the type of `xpVar` now actually contains the XML type `Node`.

For constructors with more than one field we can do the same trick, only translating to tuples instead of single values:

```
xpStack :: PU t Widget
xpStack = xpWrap (\(Stack id children) -> (id, children))
                 (\(id, children) -> Stack id children)
                 ...
```

This means that we must now deal with a pair of type `(String, [Widget])`. For this we use `xpElem` which also handles child elements, to handle the `children` as elements. We can now finish defining `xpStack`:

```
xpStack :: PU [Node] Widget
xpStack = xpWrap (\(id, children)      -> Stack id children)
                 (\(Stack id children) -> (id, children))
                 $ xpElem "Stack"
                   (xpAttr "id" xpString)
                   (xpList xpVar)
```

As soon as more than one attribute or element we use another convenience pickler `xpPair`[5] which combines two picklers into a pickler that handles the different parts of the pair.

We can thus use `xpPair xpString (xpList xpVar)` where the first argument `xpString` handles the first element of the pair and `xpList xpVar` handles the second element. The function `xpList p` does nothing else than during unpickling applying `p` to each single element until it does not return anything. During pickling it applies `p` to each element in the list.

---

[4] `xpElem :: Name -> PU [Attribute] a -> PU [Node] n -> PU [Node] (a,n)`

[5] `xpPair :: PU [a] b1 -> PU [a] b2 -> PU [a] (b1, b2)`

We can immediately see how this scales to more than two fields in a constructor. We can create bigger tuples and handle them with `xp3Tuple`, `xp4Tuple`, ... functions where `xpPair` is used . Note that `xp2Tuple` is the same as `xpPair` and `xp1Tuple` equals `id`.

This approach works for our simple example. But for more interesting nested widgets, we need a pickler for the whole data type. To create this `xpWidget` we can use `xpAlt`[6], which takes a selector function that maps the different constructors to a number corresponding to the index of pickler to use in the second list argument.

```
xpWidget :: PU [Node] Widget
xpWidget = xpAlt tag ps
  where tag w = case w of
          Var _ -> 0
          Stack _ _ -> 1
        ps = [xpVar, xpStack]
```

Now we can change the definition of `xpStack` to support all `Widget`s as children.

```
xpStack :: PU [Node] Widget
xpStack = xpWrap (\(id, children)      -> Stack id children)
                 (\(Stack id children) -> (id, children))
                 $ xpElem "Stack"
                   (xpAttr "id" xpString)
                   (xpList xpWidget)
```

We can see that this method can easily be expanded to support more Pareci widgets. It is also clear that both the `Widget` data type and the pickler implementation is trivial to implement, given the widgets and their properties. This pickler can be generated using template haskell given the `Widget` type.

One drawback of this pickler is that it only supports some specific widget properties in either attribute-wise or element-wise specification, not both specifications. Also real element mode for arbitrary properties is not supported, only the logical child, if specified, is allowed. The parsing of the xml only works in a specific way and is not generally useful for real programs where both property values specifications are used interchangeably. The next section describes a pickler for the generalised widget that does support both ways of setting property values.

3.2.2. A pickler for the generalised widget type

For the generalised `WidgetInstance` in section 3.1.2 we can generate picklers without depending on Template Haskell.

Since we are translating the XML-tree into an `UnparsedWidget` we only need to define the pickler for that specific type: `PU [Node] UnparsedWidget`.

[6] `xpAlt :: (a -> Int) -> [PU t a] -> PU t a`

Since we are not hindered by using the exact widget constructor, we can use the provided `xpElemWithName`[7] method, which, given an attribute and node pickler, gives us a triple with the widget (XML-)name, attributes, and elements. Wrapping this using `xpWrap` gives us the possibility to match the XML-name with a widget name from a widget, to combine the properties found in the attributes and elements, and to construct an `UnparsedWidget` instance. When combining the two sets of properties we can immediately check for duplicate properties and report this.

The other way around when producing the XML we have to supply the widget name, a list of properties which we want to output as attributes and a list of properties that we want to output as elements. Since we know we are dealing with `UPropVal` we can ignore all `NotSet` since we do not need it in the output, output all `Unparsed` information as attributes (, since we know that they can only be strings), and output all `UResolved` widgets as child elements using element property syntax.

In the actual implementation we use a modified version of `xpElemWithName` that passes the definition of our widgets to the picklers used for the attributes and elements: `xpElemWithWidgetDef`[8].

```
xpUnparsedWidget :: WidgetDefinitions -> PU [Node] UnparsedWidget
xpUnparsedWidget wds =
  xpWrap
    (\(name, props, props')
       -> WI (getWidgetDefFromWds wds name) (joinProps props props'))
    (\(WI wd (M.toList -> props))
        -> ((createName $ name wd)
           , filterPropsForAttr props
           , filterPropsForElem props))
    (xpElemWithWidgetDef wds
     (xpUPropsAttr)
     (xpUPropsElem wds)
    )
```

The function `getWidgetDefFromWds` takes care of looking up the widget definition given a widget name. `joinProps` joins the sets of found properties and handles possible duplicate properties. The `filterPropsFor` functions filter the PropVals as described above. Some extra care has to be taken such that the Pareci namespaces are respected. One can for example declare a `Goto` action widget as `Widget_Action_-Goto` of `a:Goto` using namespace `urn:Widget_Action`. If the logical child is set for a widget, we have to allow this as child element as well.

---

[7] `xpElemWithName :: PU [Attribute] a -> PU [Node] n -> PU [Node] (Name,a,n)`

[8] `xpElemWithWidgetDef :: WidgetDefinitions -> (WidgetDefinition -> PU [Attribute] a) -> (WidgetDefinition -> PU [Node] n) -> PU [Node] (Name,a,n)`

3.3. Expression and Binding Syntax

Many values in Pareci are constructed using property values. Property values can be divided into two parts: the expressions and the bindings. Expressions contain arithmetic operators, logical operators and plain numbers, strings and booleans and binding syntax refers to dynamically bounded variables.

Whether a string is a binding or expression depends on the first character. If it is an equals (=) sign it is a expression, otherwise it can be a simple string or a binding. A binding can be recognised by its surrounding curly brackets (*{ }*).

The implementation of the property value parser is made using the *uu-parsinglib* [Swi09].

The grammar and data types for the property values can be found in appendix B

3.4. Combining the Pickler and Property Value parser

The XML picklers and the expression syntax parser can be combined to check both the content and the structure of a Pareci page or application. The next step is to actually validate the semantics of the page to see if the pages make sense, which we will not do here.

To summarise we have bidirectional functions to translate between the Page XML, the `UnparsedWidget` and `ParsedWidget`. This is illustrated in fig. 3.1.

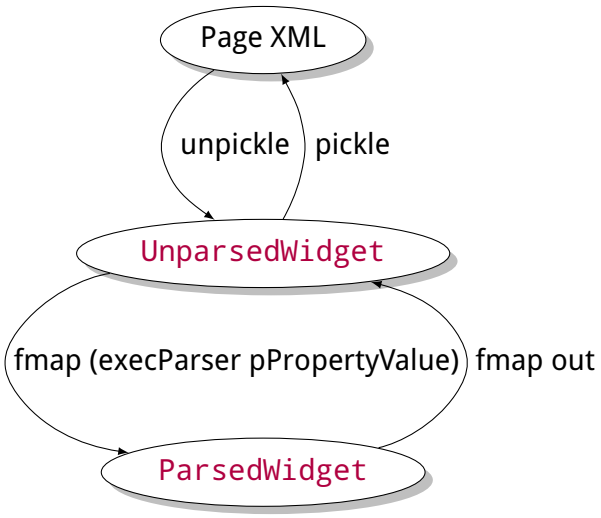


Figure 3.1.: Bidirectional translation from XML type to Analysis types

The pickler defined in section 3.2.2 provides the `pickle` and `unpickle` functionalily. The parser and `Functor` implementation of `WidgetInstance` makes sure we can run the parser from section 3.3 to translate from `UnparsedWidget` to `ParsedWidget`. The translation back is done by simply outputting the parsed expressions and bindings

to plain string using the `out` function. Note that this is simplified a bit, since `PPropVal` also contains widget instances and not just strings.

The next chapter discusses how to describe an analysis of a Pareci program as a Monotone Framework and how to perform these analyses.

4. Data Flow Analysis

To be able to analyse Pareci we will need to have some kind of analysis methodology. We will use data flow analysis. This is based on analysing the control flow of the analysed program. We will first look at what data flow analysis means by discussing it using some simple examples and how we can use a monotone framework formalisation and a worklist algorithm to solve them. Then we will focus on applying these ideas to Pareci.

4.1. Program Analysis

Before we delve into the analysis of Pareci, let us first look at program analysis in a more generic way. In essence with program analysis we can do static compile-time analysis, which will result in fewer run-time bugs and allow for optimisations. This is done by analysing all program statements and combining knowledge about them from different parts of the program into information that can be used to filter out for example redundant computations and undefined variables.

One of the most important parts of data flow analysis is the control flow graph. This control is a graph that represents the flow of a program execution. Each node corresponds to a program point which we will represent as $[\mathrm{p}]^{\mathrm{l}}$ where p is the program point and l is a unique label. Basically each program point corresponds to a statement in the program source code or a point on which the control flow of the program can diverge or converge. An example for the last case is a function call. Edges between nodes represent a flow of control during program execution.

We will discuss some useful classical program analyses below as defined in [NNH04], which the analyses on Pareci are based on.

4.1.1. Classical Program Analyses

Reaching Definitions

Reaching Definitions is an analysis that determines which relevant assignments have been made at points in program executions. This allows us to determine whether there are unnecessary assignments. For example see the following piece of code, where $:=$ denotes is an assignment and x and y are variables.

```
y := 1
y := 2
x := y
```

We can easily see that the first assignment y := 1 is redundant, because y immediately gets reassigned another value in y := 2. Reaching definitions analysis can detect these kind of redundant assignments.

This analysis can be done by following the program flow and keeping track of which assignments are done where and overwriting them when another assignment is done.

Live Variables

A variable is live if it holds a value that may be needed in the future, i.e. a variable may potentially be read before it is reassigned.

Live Variable analysis is used to determine if there is code that is executed but which has no observable effect on the program's result. It can also detect if variables are being used before they are defined.

4.1.2. Analysis results

The result of an analysis is a mapping from program points to a relevant value. To generalise this, we restrict the values for each result set to the elements of a *complete lattice* [Wil12, NNH04, Fri11]. A lattice is a partially ordered set, which captures the intuitive concept of an ordering. This ordering is defined for each lattice L with a reflexive, anti-symmetric, and transitive binary relation $\sqsubseteq$. L also has a *least upper bound* or *join* ($\sqcup$) and a *greatest lower bound* or *meet* ($\sqcap$). Its greatest element is called *top* ($\top$) and its least element is called *bottom* ($\bot$), meaning that $\forall a \in L. (\bot \sqsubseteq a \sqsubseteq \top)$.

Elements of a lattice are used to represent analysis results. Given a lattice L, for each program point we have two kinds of values: a context value $L_\circ$. The effect value is thus always dependent on the context value and is calculated by a transfer function. The calculated effect values are propagated to the context values of the successors program points in the flow graph.

Intuitively a lattice's bottom is the most amount of information, increasing by each larger element and top is the least amount of information.

4.2. Monotone Frameworks

All analyses described above can be formalised as instances of Monotone Frameworks. Using a similar framework to represent the analyses, we can make use of generic algorithms to solve the data flow equations.

For all of the analysis variants [NNH04] that they have this overall pattern.

$$\mathit{Analysis}_\circ(l) = \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{\mathit{Analysis}_\bullet(l') \,\|\, (l', l) \in F\} & \text{otherwise} \end{cases}$$

$$\mathit{Analysis}_\bullet(l) = f_l(\mathit{Analysis}_\circ(l))$$

where $\bigsqcup$ is the join of the lattice L, F is the control flow, ι is the initial analysis value and f_l is the transfer function associated with program point l. (a, b) is used to denote an edge from node a to node b.

We have two kinds of analysis directions, *forwards* and *backward*. The forward analysis means that the program control flow ($flow(S_*)$) is used as the program executes; the backwards analysis has exactly the program control flow backwards ($flow^R(S_*)$), which can be easily obtained by reversing each edge in the directed flow control graph. S_* denotes all the statements in the analysed program. Typically extremal nodes are each others duals for directions.

Transfer function $f_l \in \mathcal{F}$, where $\mathcal{F}$ is the set of monotone functions from L to L including the identity function. f_l is a transfer function that computes the effects of program point l given the context of l. It is monotone in the sense that $l \sqsubseteq l' \Rightarrow f_l(l) \sqsubseteq f_l(l')$, meaning that an increase in knowledge in the input l may not lead to a decrease in the knowledge of the output $f_l(l)$.

L must be a complete lattice that satisfies the Ascending Chain Condition, meaning that each ascending chain $l_1 \sqsubseteq l_2 \sqsubseteq l_3 \ldots$ eventually stabilises, $\exists n.(l_n = l_{n+1} = \ldots)$. Because we choose our lattices finite (i.e. only picking the actual variables used in our program instead of all possible variables), this is the case.

To make an instance of a monotone framework [NNH04] we thus need:

- a complete lattice L
- a function space $\mathcal{F}$
- a finite flow F, typically forward $flow(S_*)$ or backward $flow^R(S_*)$
- a set of *extremal labels* E, typically the initial or final program points
- an *extremal value* $i \in L$, the starting value of the extremal program points
- a mapping f which maps the program points in F and E to transfer functions in $\mathcal{F}$

4.2.1. Worklist algorithm

To compute the analysis result we must "solve" the monotone framework. This can be done using a maximal fixed point algorithm, in our case a worklist algorithm based on the Maximal Fixed Point solution in Section 2.4 of [NNH04].

The input to the algorithm is an instance of a Monotone Framework ($\langle L, \mathcal{F}, F, E, \iota, f \rangle$).

The output is a stable mapping from program points to analysis result context ($Analysis_\circ$) and effect values ($Analysis_\bullet$).

A formal description can be found in algorithm 1. We can roughly divide it into three steps: initialisation, iteration and presenting.

The analysis result for each program point is initialised bottom, or if concerning an extremal program point as ι. The algorithm adds all edges in the control flow graph to a worklist W. Then it will continue until W is empty. With each item it will check if it yields stronger information (using the ordering relation $\sqsubseteq$) and if so add it to the result.

Algorithm 1 Worklist algorithm

function MFP($L, \mathcal{F}, F, E, \iota, f$)

$Analysis[l] \leftarrow \begin{cases} \iota & \text{for } l \in E \\ \bot_L & \text{otherwise} \end{cases}$ ▷ Initialisation

$W \leftarrow F$ ▷ Initialise worklist W

while W not empty **do** ▷ Iteration

$(l, l') \leftarrow head(W)$ ▷ Pop the next item from the worklist

$W \leftarrow tail(W)$ ▷ Update the worklist

if $f_l(Analysis[l]) \not\sqsubseteq Analysis[l']$ **then**

▷ Applying the transfer function yields new information

$Analysis[l'] \leftarrow Analysis[l'] \sqcup f_l(Analysis[l])$ ▷ Merge the analysis result

for all l'' with $(l', l'') \in F$ **do**

▷ Add edges for which the result is influenced by l'' to the worklist

$W \leftarrow (l', l'') : W$

$Analysis_\circ \leftarrow Analysis$ ▷ Presenting

$Analysis_\bullet \leftarrow map\ f_l\ Analysis$

return $(Analysis_\circ, Analysis_\bullet)$

Any edges that are influenced by this update are again added to W. Since our lattice adheres to the Ascending Chain Condition the worklist will eventually become empty since no weaker results are encountered.

The Haskell implementation used for the Pareci analysis can be found in appendix C.

Optimisation

To optimise the running time of the algorithm discussed above we can make sure the worklist is initialised in an order for which we know that it will result in decreasing the number of edges re-added to the worklist.

In a forward analysis it is fastest if all predecessors of a program point have already been processed before the point itself. The iteration will then contain the latest information. This corresponds with a reverse postorder traversal of the context flow graph starting at the entry point of the application. Reverse postorder iteration visits a node before any of its successors have been visited. For backwards analysis we want to visit a node after any of its successors have been visited, this is done by postorder traversal.

We replace the initialisation step of the worklist by the following statement to make the worklist algorithm follow the analysis direction better, in order to be more efficient.

$$W \leftarrow \begin{cases} \text{reverse postorder traversal of } F & \text{if forward analysis} \\ \text{reversed edges of postorder traversal of } F & \text{if backwards analysis} \end{cases}$$

For the forwards analysis this is implemented for the control flow graph by doing a postorder traversal and skipping nodes that have already been visited. The last step is

to reverse the result list. A backward analysis uses the same postorder traversal and swaps the edge direction.

4.2.2. Interprocedural data flow analysis

Many programming languages support some kind of functions or method calls. This is something the analysis should also deal with. A naïve solution would be to insert a copy of the program points of the function everywhere the function is called, but this is a not a viable solution since it would increase the number of program point dramatically and in languages supporting recursion infinitely much. We have to do something smarter.

We follow the approach found in [NNH04] and introduce two program points for each procedure call, corresponding to the call and the return of the procedure. The programs flow is represented by connecting the call point to the first program point in the function and the last point in the function to the return point.

To drawback of this approach is that the result of all function calls is combined in each return, resulting in a loss of specificity. To deal with this we introduce the concept op context-sensitive analysis. This means that we do not only have an analysis result in terms of lattice L, but also specified by some context Δ, which is used to make the distinction between different function calls. We lift our analysis result to a new lattice $\widehat{L} : \Delta \to L$. This means that we also have a lifted transfer function $\widehat{f_l} : \widehat{L} \to \widehat{L}$; this can be easily implemented by applying f_l to each lattice value separately, effectively keeping the different contexts separate: $\widehat{f_l}(\widehat{l})(\delta) = f_l\left(\widehat{l}(\delta)\right)$.

To handle the updating of Δ we need a way to represent the context changes for relevant the program points. For an interprocedural analysis this are the calling of procedures and the returning of procedures.

In fig. 4.1 we find a representation of the call and return process. The nodes represent program point $[S]^l$ for statements S and with label l. The edges contain the analysis result values. The entry and exit of the procedure are represented respectively by $[\mathrm{Entry}]^{l_n}$ and $[\mathrm{Exit}]^{l_x}$. The $[\mathrm{Call}]^{l_c}$ program point represents the call to the procedure and the $[\mathrm{Return}]^{l_r}$ program point the return of the procedure. The handling of the incoming analysis result $\widehat{l}$ is handled by the transfer function $\widehat{f^1_{l_c}}$ for the call program point. This function also handles the analysis context changes corresponding with the procedure call The analysis result from point l_x, corresponding with the exit of the function is handled by $\widehat{f^{2B}_{l_c,l_r}}$, which also handles the context changes corresponding with the procedure return. The transfer function used for return program point is $\widehat{f^2_{l_r}}\left(\widehat{l}\right)\left(\widehat{l'}\right)$, which consists of joining the results coming from the exit point and the call node: $\widehat{f^{2A}_{l_c,l_r}}\left(\widehat{l}\right) \sqcup \widehat{f^{2B}_{l_c,l_r}}\left(\widehat{l'}\right)$. The result coming the call point, can also be transfered, which handled by $\widehat{f^{2A}_{l_r}}\left(\widehat{l'}\right)$.

By using lifted lattices and transfer functions, we only have to make a small adjustment to our implementation of the worklist algorithm. The function $\widehat{f^2_{l_c,l_r}}$ is a binary

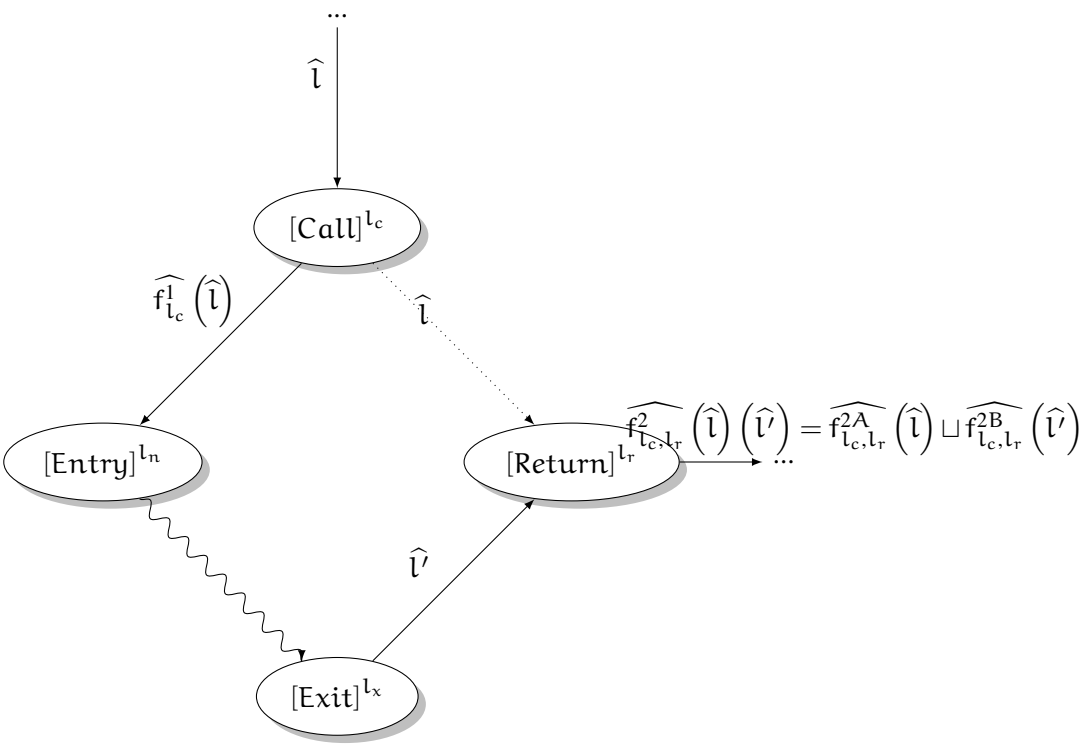


Figure 4.1.: Interprocedural flow and transfer function

transfer function, which needs to be called correctly by the worklist algorithm. To be able to discriminate in using the original binary transfer function and the newly defined unary transfer function, we use a Monotone Framework variant called *Embellished* Monotone Frameworks, which contains a notion of *inter-flow*, which is the set of tuples containing $\langle l_c, l_n, l_x, l_r \rangle$ containing the labels for each function call. The worklist algorithm can use the binary transfer function whenever the (l_x, l_r) edge is encountered in the worklist and the unary transfer function in all other cases. The l_c label can be used to retrieve the $\widehat{l}$ from the analysis result set. The context changes get handled by the lifted transfer function and the lifted lattice adheres to the same properties as the original lattice. The new Embellished Monotone Framework instance is $\left\langle \widehat{L}, \widehat{\mathcal{F}}, F, IF, E, \widehat{\iota}, \widehat{f_l} \right\rangle$, where IF is the inter-flow.

The adaptation of the worklist algorithm based on Embellished monotone frameworks can be found in algorithm 2 and the Haskell implementation in appendix C.2.

Algorithm 2 Worklist algorithm

function MFP$(L, \mathcal{F}, F, IF, E, \iota, f)$

$Analysis[l] \leftarrow \begin{cases} \iota & \text{for } l \in E \\ \bot_L & \text{otherwise} \end{cases}$ ▷ Initialisation

$W \leftarrow F$ ▷ Initialise worklist W

while W not empty **do** ▷ Iteration

$(l, l') \leftarrow head(W)$ ▷ Pop the next item from the worklist

$W \leftarrow tail(W)$ ▷ Update the worklist

$newAna \leftarrow \begin{cases} f^2_{l_c,l_r}(Analysis[l]) & \exists (l_c, l_n, l_x, l_r) \in IF.(l \equiv l_r) \\ f^1_l(Analysis[l]) & \text{otherwise} \end{cases}$

if $newAna \not\sqsubseteq Analysis[l']$ **then**

▷ Applying the transfer function yields new information

$Analysis[l'] \leftarrow Analysis[l'] \sqcup newAna$ ▷ Merge the analysis result

for all l'' with $(l', l'') \in F$ **do**

▷ Add edges for which the result is influenced by l'' to the worklist

$W \leftarrow (l', l'') : W$

$Analysis_\circ \leftarrow Analysis$ ▷ Presenting

$Analysis_\bullet \leftarrow map\ f_l\ Analysis$

return $(Analysis_\circ, Analysis_\bullet)$

This context can be chosen as any property for which we want to differentiate the analysis by.

For an interprocedural analysis we want to distinguish between the different function calls, therefore we pick the Δ as *call strings*. A call string is a list of identifiers of which procedure calls are previously made. A good identifier would be the label of the corresponding call program point. In this case the lifted unary transfer function $\widehat{f^l_l}$ can use the transfer function for L and keep the context identical for all program points

that are not a call or return. The binary transfer function $\widehat{f}^2_l$ updates the call string: for calls we add the call label l_c to the context values and for return functions we remove l_c from the call string again. This is the case for forwards analyses. Backward analyses have the function for call and return program points swapped. To make sure that the call string does not increase infinitely in case of a recursive function call, we can limit the call string length.

Using context-sensitive analysis only relevant information from the corresponding call site is available after the return is handled. Any information from different calls to the same function are ignored by the transfer function of the return point, while only needing one representation of the function flow in the control flow graph.

5. Analysing Pareci

5.1. Pareci as a Monotone Framework Instance

To be able to do data flow analysis of Pareci programs, we need to describe a way to turn a Pareci application into a Monotone Framework instance. An important part of the instance is the program flow. Since Pareci pages are realtime applications dependent on user input, this can be a hard task.

This sections defines how we can create a Monotone Framwork instance for Pareci. One of the most important aspects of data flow analysis is the data flow. Since Pareci is GUI-driven and uses events to process user action, we have to model this in the flow. We look at program analysis for other GUI-driven programming environments and how we can use those ideas to define the program flow for Pareci.

5.1.1. GUI and events

There are some similarities between Android and Pareci. Both use XML-based declarative layout files and are event-based. Payet en Spoto [PS11] describe how to optimise their static analysis tool Julia to deal with Android applications. In Android applications the XML layout files are initialised using Java reflection and thus declared at runtime. Their model has support for adding the required properties, fields and view objects to the analysis context, thereby making them available for analysis. In this case the initialisation is driven by the Java-code. To do something similar the Pareci framework implementation would also need to be analysed. Since this is out of the scope of this thesis, a similar approach can be emulated in a step before the analysis.

An initialisation of the analysed Pareci page could be used to make sure that all the static components of the page are available. This means the extremal values for the analysis might not be the bottom of the analysis lattice, but one with more information available. For Android most events can be called from outside the program, so Payet en Spoto's [PS11] solution is to add these as entry points to the application. In Pareci this is not the case, so we do not have to deal with this issue. Events can thus be handled by doing the initialisation step as described above.

Staiger [Sta07] describes a way to detect which elements of the program are part of the GUI and how to handle events in Android program analysis. The former is not important for analysing Pareci, since we already know which widgets are GUI widgets, but the latter aspect is interesting. They deal with events by adding a main loop to the program, in order to let the analysis find out which relevant events can happen. We apply this strategy to Pareci to work with the actions that a user can do while

interacting with the Pareci application. The main-loop program point is inserted after the page initialisation steps are completed.

A side note here is that Pareci can handle multiple requests at once and also has options to delay specific events (or `Actions`) which would result in parallel handling. In this thesis we only look at sequential handling of (user) events. This simplified view makes the analysis less complex and thus more understandable and better performing. An alternative approach here to deal with multiple requests is to make it possible to break out of and action and later on return to finish that action. This can be easily implemented by respectively inserting additional return and calls during the main program loop as discussed before.

5.1.2. Pareci flow

Page Outline

We define the flow of a Pareci page in two parts. First we initialise the scope of the page, meaning all UI elements, variables and resources. As a rule of thumb we can use all widgets that inherit from `Widget_Widget` as UI widgets. More specifically , we can check the information/types of the widget properties where children of `Widget_Widget` are allowed. Then we insert a main loop in which the (user) events are linked.

In fig. 5.1 we can see the basic layout of a Pareci page flow. When the page is accessed we start in program point Start, then first the UI widgets are added to the flow (in $\mathrm{UIWidgets}$). Subsequently we enter a program point called $\mathrm{EventLoop}$ at which point events can occur. An event triggers an action execution. After an action is finished, the control flow is returned to the $\mathrm{EventLoop}$. From there another action can be triggered or the execution can finish by following the flow to End.

Note that since this is a flow analysis, there are no problems here with the infinite loop, because we have chosen an analysis lattice which conforms to the Ascending Chain Condition as discussed in section 4.2.

The initialisation of the UI widgets is sequential. We traverse the Pareci page in a depth first order, creating flow for every child widget in the page `message`, `resources` and `content` property values. The next step is to add the events to the flow. After creating the $\mathrm{EventLoop}$ node, we can make flow to and from all possible callable actions. For `ActionLists` we can do the same sequential composition as for the child widgets of the page. All actions in the list can be connected in execution order.

More generally, there are two ways of handling widgets. One for those with children, such as `Page` and `Stack` and for those without such as `LinkButton` and `Debug`. Those without are the easiest to handle. They have exactly one node and have flow from their predecessor, and flow to the next in line. Basically their flow is dependent on what comes before and after. Widgets with children have a node for themselves and a flow to their children and between them.

Each widget has a program point associated with it in the data flow. We treat widgets as single entities, which in essence means that they have single effect on the analysis result. We define a transfer function that handles all the properties and specifics of

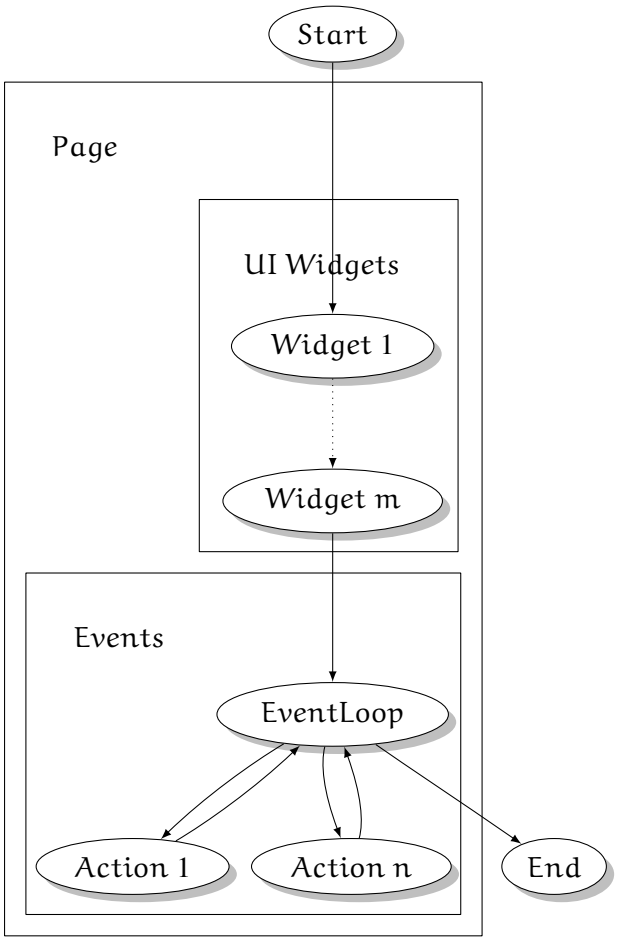


Figure 5.1.: Schematic Pareci Page Flow

the widget in section 5.2.2. Most widgets have no flow, unless they can contain child widgets which have special meaning, such as the children of the `Page`, `ActionList` and `Trigger` widgets.

Specification of Pareci flow

More formally we can define the above approach as a support function for Data Flow Analysis. Let us first define a widget w without children $[w]^l$ with label l and a widget $[w_c]^l$ with children $c_1 \ldots c_n$. .. Any Haskell code in this section is based on our first widget definition attempt using a constructor for each widget, found in section 3.1.1.

To make this easier to describe we first introduce $init$ and $final$ functions, where $init$ gives us the initial label of a statement and $final$ the set of last labels.

$$
\begin{aligned}
init &: ParsedWidget \to Label \\
init([w]^l) &= l \\
init([w_c]^l) &= l
\end{aligned}
$$

$$
\begin{aligned}
final &: ParsedWidget \to \mathcal{P}(Label) \\
final([w]^l) &= \{l\} \\
final([w_c]^l) &= final(c_n)
\end{aligned}
$$

$$
\begin{aligned}
flow &: ParsedWidget \to \mathcal{P}(Label \times Label) \\
flow([w]^l) &= \emptyset \\
flow([w_c]^l) &= \{(l, init(c_1))\} \\
&\quad \cup \left(\bigcup_{c_i \in c \setminus c_n} (\{(f_i, init(c_{i+1})) \| f_i \in final(c_i)\}) \right) \\
&\quad \cup flow(c_1) \cup \ldots \cup flow(c_n)
\end{aligned}
$$

The flow for $flow([w_c]^l)$ connects the current widget label l to the inital label of its first child. Also the possible multi-element results of $final$ of each child is connected to the next child by the second line of $flow([w_c]^l)$. The flow of this children is also added. Because we defined the $final$ of the w_c as the $final$ if the last child the widget itself is connected by its possible parent. The widget itself has no flow and its effects are dealt with in the transfer function. Some specific kinds of widgets do manipulate the flow and are discussed below at the end of section 5.1.2.

When actually implementing this we use an attribute grammar where the list of

children c is a separate terminal. The $flow$ function becomes:

$$\begin{aligned} flow([w_c]^l) =& \{(l, init(c))\} \\ & \cup flow(c) \\ flow(c) = & \bigcup_{c_i \in c \setminus c_n} (\{(f_i, init(c_{i+1})) \| f_i \in final(c_i)\}) \\ & \cup flow(c_1) \cup \ldots \cup flow(c_n) \end{aligned}$$

where for the list of children c:

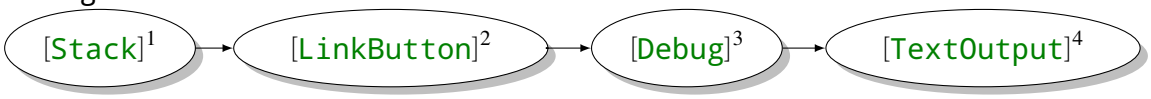
$$\begin{aligned} init\left(\left\langle [c_1]^{l_1}, \ldots, [c_n]^{l_n} \right\rangle\right) =& l_1 \\ final\left(\left\langle [c_1]^{l_1}, \ldots, , [c_n]^{l_n} \right\rangle\right) =& \{l_n\} \end{aligned}$$

The functions $init(c)$ and $final(c)$ return respectively the label of the first and last child of c.

Basically we are creating a chain of widgets in sequential order and connecting their $final$s and $init$s. For example consider the following Pareci page fragment to illustrate the flow creation for children:

```
<Stack>
   <LinkButton />
   <Debug />
   <TextOutput />
</Stack>
```

We see widget `Stack` with children $\langle$`LinkButton`, `Debug`, `TextOutput`$\rangle$. The corresponding flow is shown below.



with $init\left([\texttt{Stack}]^1\right) = 1$ and $final\left([\texttt{Stack}]^1\right) = final(\text{children of } \texttt{Stack}) = \{4\}$.

This way of defining flow is implemented using the Utrecht University Attribute Grammar System [DS05]. We traverse over the Pareci widget tree and whenever a widget has child widgets we add them.

The next step is to deal with the events. For this we have to create a special node which represents the $\mathrm{EventLoop}$. Remember that we have a `ParsedWidget` type for representing the Pareci widgets. For the UI elements above, this was enough to define the flow, but if we are introducing new analysis nodes such as $\mathrm{EventLoop}$ we need to extend it. Therefore we wrap `ParsedWidget` in `ProgramPoint` which can contain an $\mathrm{EventLoop}$ node or a Widget.

```
data ProgramPoint = WrappedW {widget :: ParsedWidget}
                  | EventLoop
```

This way we can represent all widgets (albeit in a wrapper) and the $\mathrm{EventLoop}$. The $\mathrm{EventLoop}$ is added to the flow by the Page widget. After all non-action widgets are added to the flow an $\mathrm{EventLoop}$ node is created with the following properties:

$$\begin{aligned} init\left([\mathrm{EventLoop}]^{l}\right) &= l \\ final\left([\mathrm{EventLoop}]^{l}\right) &= \{l\} \\ flow\left([\mathrm{EventLoop}]^{l}\right) &= \text{the set of edges of } l \text{ to the possible triggered actions} \end{aligned}$$

We discuss the connecting of actions calls below. This leads to the following flow for a `Page`:

$$\begin{aligned} init\left([\mathtt{Page}]^{l}\right) &= l \\ final\left([\mathtt{Page}]^{l}\right) &= l_{\mathrm{EventLoop}} \\ flow\left([\mathtt{Page}]^{l}\right) &= flow\left(w_{c_{\mathtt{Page}}}\right) \cup \left\{\left(f, l_{\mathrm{EventLoop}}\right) \| f \in final\left(c_{\mathtt{Page}}\right)\right\} \end{aligned}$$

where $w_{c_{\mathtt{Page}}}$ is `Page` as a widget with children as above. The children here are the `message`, `resources` and `content` properties of the `Page`. $l_{\mathrm{EventLoop}}$ is the label of the event loop that is instantiated during the calculation of the flow of `Page`.

The flow of the created `EventLoop` is now extended by adding also the flow used for the actions. To get all the possible events, we create another attribute which stores the events encountered when traversing the `Page`. This consists of basically all Action Widgets on the `Page`.

The last step is to wrap the whole analysis with a `Start` and `End` node. This is to make sure we have a single start and end point for the analysis. We add those constructors to the `ProgramPoint` data type.

```
data ProgramPoint = [...]
                  | Start
                  | End
```

We also make sure that when the analysis is started they are added to the flow. To do this the whole analysed program gets wrapped in a `Top` constructor to handle this part.

```
data Top a = Top {unTop :: ProgramPoint}
```

`Top` only takes over the flow if its wrappee and connects Start to its $init$ and its $finals$ to End.

$$\begin{aligned} flow(\mathtt{Top}) = &flow(unTop) \\ &\cup (l_{\mathrm{Start}}, init(unTop)) \\ &\cup \{(f, l_{\mathrm{End}}) \| f \in final(unTop)\} \end{aligned}$$

$unTop$ refers to the wrapped Widget and l_{Start} and l_{End} refer to the created Start and End program points.

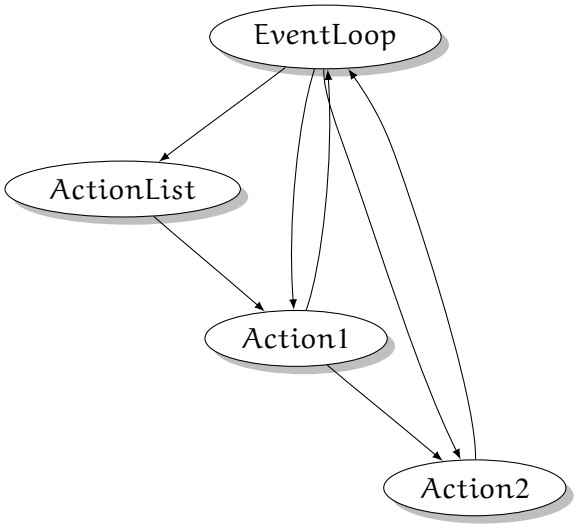


Figure 5.2.: Too many paths in an action list

Drawbacks

There are a couple of drawbacks to the approach sketched above.

This approach includes all possible program execution paths of a Pareci program, but it is not what Pareci does and will make the analysis result imprecise. It produces too many paths, for example when calling actions.

Consider fig. 5.2; we can either reach $\mathrm{Action1}$ directly or by calling $\mathrm{ActionList}$. This is no problem in itself, but we also get a situation now where we have the following path $\{\mathrm{EventLoop}, \mathrm{ActionList}, \mathrm{Action1}, \mathrm{EventLoop}\}$ and thus skipping $\mathrm{Action2}$, event though it is part of the $\mathrm{ActionList}$. The path $\{\mathrm{EventLoop}, \mathrm{Action1}, \mathrm{Action2}, \mathrm{EventLoop}\}$ has the same issue. This means that there are possible execution paths that can never occur in a real program and thus only make the analysis stricter than it needs to be.

We handle this by treating the triggering of actions as functions calls and using the interprocedural analysis technique as discussed in section 4.2.2. For each call to an action we introduce a `Call` and `Return` corresponding with a call site to the action. To denote the end of an execution of the widget we also introduce `Exit`. We expand the `ProgramPoint` type for this:

```
data ProgramPoint = [...]
                  | Call   {ident :: Ident}
                  | Return {ident :: Ident}
                  | Exit   {ident :: Ident}
```

This means that the flow of each action contains a program point for the action itself, its children if any, and an `Exit` node. We require the added program points to have a identifier (`ident`) corresponding with the action.

The `Call` and `Return` are only created and connected if an action is actually callable and referenced in an event. An action is callable if it has an `id`[9]. To determine which

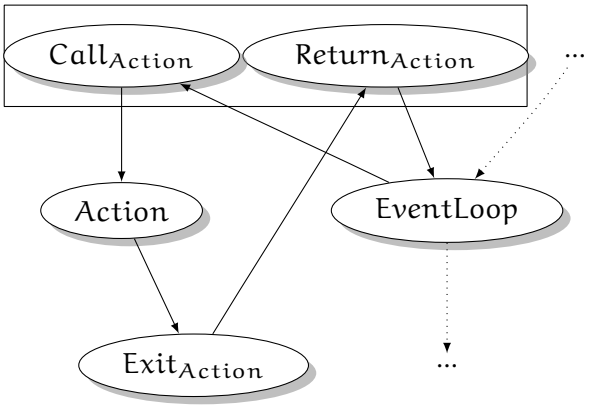


Figure 5.3.: Action call and return flow

actions are called, there is a pass over all the widgets, in which we collect all the `id`'s referenced in event properties such as the `onClick` property[10]. There are two special cases: `ActionList`s and `Trigger`s. In the case of an `ActionList`, calls are created to every child and connected sequentially. A `Trigger`can trigger an action and therefore create a `Call` and `Return` connected to the triggered action.

A simple action call's flow diagram can be seen in fig. 5.3.

In fig. 5.4 we can see an example of a fragment of the resulting flow graph containing an `ActionList` with a `Log` and a `Trigger` which calls the `Log` again. The `ActionList` is called by some event on the page and therefore called via the `EventLoop`. When we follow the flow from the $\mathrm{EventLoop}$, we can see that it correctly goed through the `ActionList`, which calls the `Log` and then we choose at $\mathrm{Exit}_{\mathtt{Log}}$ which path to Return to pick. This is not a problem, because it is handled by the interprocedural analysis. The Return corresponding with call from the `ActionList` (denoted with box), leads us to the call of the `Trigger`, which in turn calls the `Log` again. In this example we have seen the handling of sequential calls by the `ActionList` and the handling of the `Trigger`[11]. This rougly corresponding with the following Pareci code fragment:

```
<a:ActionList id="al">
  <a:Log id="log" />
  <a:Trigger target="log" />
```

[9] Note that this is actually a simplification, because actions can also be called by referencing it in another way than ID, for example by referencing its location in a list. Given a `Page` with `id` `pageId` a valid binding to the first action of the page would be `{#pageId.actions.1}`. For this prototype we do not support this, because it is rarily used.

[10] Here it is also possible to call an action not with a direct `id` but by using an expression which evaluates to a valid id. This is not supported at the moment.

[11] The actual implementation also handles `Var` as children of a `Trigger`. This is used for assigning new values to `Var`s on page the target action is on. Since our analysis prototype does not support calls to other pages, we can safely implement this by inserting extra `Assign`s into the flow replacing each `Var` and handling them accordingly.

```
</a:Actionlist>
[..]
<LinkButton onclick="{#al}" />
```

Dependency Analysis

When naïvely building up the program flow top-down from a page definition, we run into problems with flow order. A widget can reference any other widget on the page. This means that every other widget can be referenced. In Pareci there is no strict order, the whole page just exists and is accessible. Bindings are resolved lazily.

To deal with this when creating the flow for a page, we can look at the dependency graph of a page. We build a dependency graph by associating with each widget which variables it provides and which variables it depends on. Next we can detect cycles in this graph by finding any strongly connected components which contain more than one widget [KL94].

If a cycle is found, we can now put out a warning to the user. The analysis can still continue, but is no longer guaranteed to be correct, since some widgets might depend on widgets defined earlier in the flow.

A topological sort of the dependency graph now results in the actual order of widgets depending on each other. To improve the flow definition we can connect this ordering in a sequential manner and add it to the program flow instead of adding them in the order as done earlier. We update the $flow$ function for Page:

$$\begin{aligned} flow\left([\text{Page}]^l\right) = &\text{topological sort of dependency graph of Page's "UI widgets"} \\ &\cup flow(\texttt{actions}) \\ &\cup \left\{(f, l_{EventLoop}) \,\|\, f \in final(c_{\text{Page}})\right\} \end{aligned}$$

where the UI widgets are the children in the Page's `message`, `resources` and `content`. The flow of the `actions` of the Page is also added.

5.1.3. Variables

Almost all analyses use some kind of representation of variables, so we need a good way to represent them for Pareci. Pareci does not have variables in a standard way, since in Pareci all bindings occur in a context as discussed in section 2.1.3. Let us start with a way to look at variables:

```
type Var = Binding
```

A Var can be represented as a Binding. For example the binding `{#person.name}` directly refers to the object (in scope) with `id person` on the page. Essentially this is a variable `#person` which stores the value of the object associated and `name` is a property on this variable and can be seen as a field of this object.

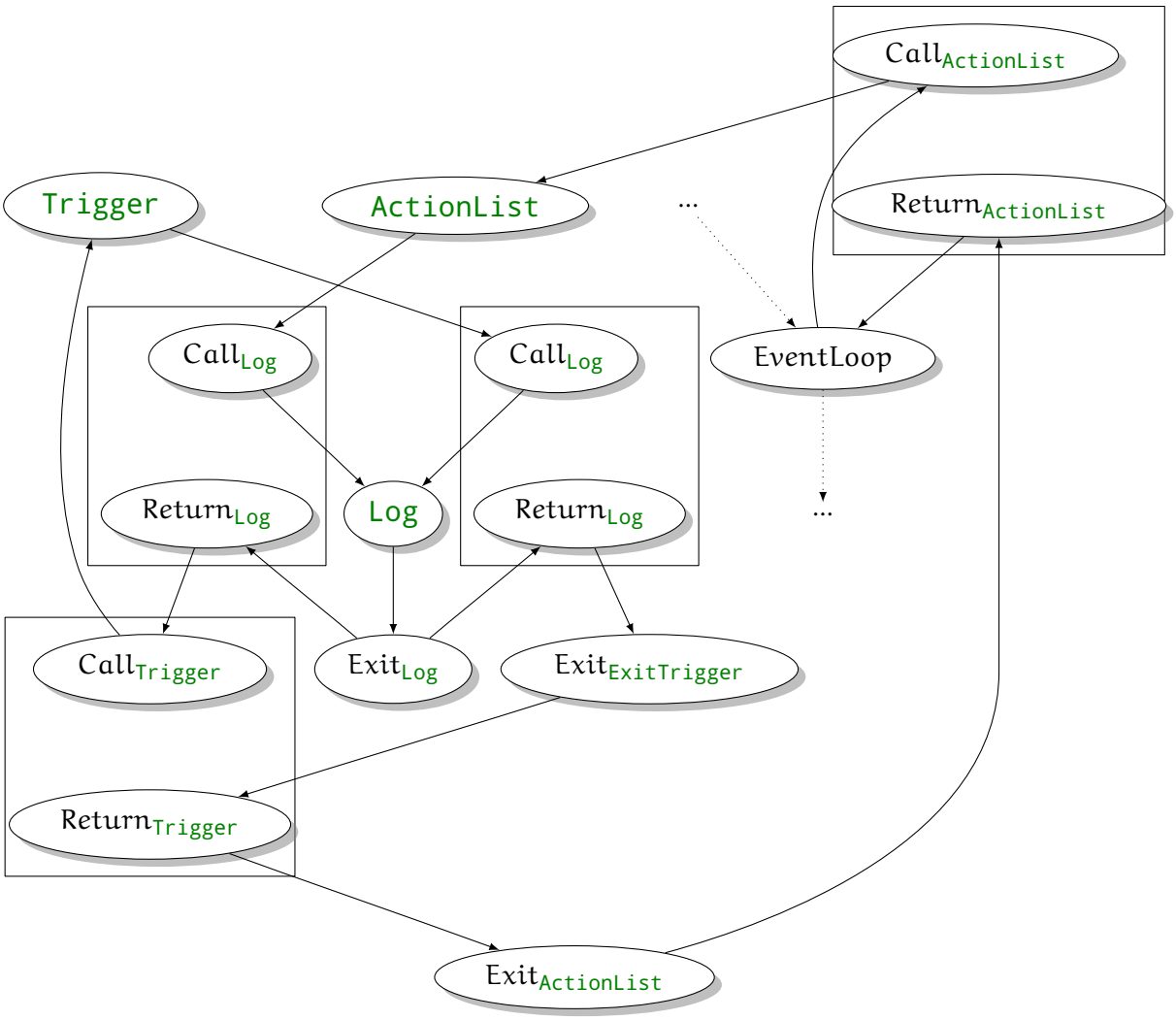


Figure 5.4.: Action flow fragment with trigger

Every binding type can be reduced to a normalised form $\widehat{b}$ as discussed before in section 2.1.3. Associated with this normal form are the `Binding` constructors as found in appendix B. This is shown in table 5.1.

Table 5.1.: Reduction to normalised form extended with `Binding` constructors

Binding syntax	**Normal form**	**Haskell constructor**
{~id}	~id	`BLocal id`
{id}	~context.id	`BContext id`
{__name.id}	~_globals.name.id	`BGlobal name id`
{^id}	~_parent.id	`BParent id`
{#id}	~_scope.id	`BScope id`

Variables are thus always relative to the current widget '~'. When the analysis results follow the flow to the next widget, we also have to make sure that the variables are still correct. Variables relative to the context must be updated to the possible new context. Scope changes also have to be taken into account.

Global variables are handled by the widget itself but should always refer to the same global object, thus for our purposes we can see this as an absolute value.

Simple properties can be handled by denoting variables as `Binding`s, since we can just introduce a new variable for every property of an object. Property `name` of variable `~person` can be introduced as `~person.name`. If method calls and expressions come into play, a type system and object model are needed to be more precise about which variables are introducible.

5.1.4. Pareci Context

Every variable is always in a context. Most of the time the context is equal to the context of the parent element in the source Page updated with the context of the current widget. Consider the following Pareci fragment:

```
<Page>
  [..]
  <Page.content>
    <Stack context="{#person.value}">
      <TextOutput id="to" context="{mother}" value="{name}" />
    </Stack>
  </Page.content>
</Page>
```

The value displayed by the `TextOutput` with `id` equal to `to` is the value of the variable corresponding with `#person.value.mother.name` for some widget in scope with `id` set to `person`. In this example it is straight forward where the variable value comes from.

The interesting case is where context is used in action calls. For example:

```
<Page>
   [..]
   <Page.actions>
     <a:ActionList id="update">
       <a:Assign id="store" field="{name}" value="{#ti.value}" />
     </a:ActionList>
   </Page.actions>
   <Page.content>
    <Stack context="{#person.value}">
      <TextInput id="ti" value="{name}" />
      <LinkButton onclick="{#update}" text="update" />
    </Stack>
  </Page.content>
</Page>
```

Action `update` is called by the `LinkButton`. Now the action is executed with the inherited Pareci context of the calling `LinkButton`. In this case the `Assign` with id `store` assigns the `name` field of the calling context `#person.value`, which results in assigning `#person.value.name` to the value that is entered in the text field of `TextInput ti`. Another call to `store` could be from another context, thus it is not static and can not be known before the analysis runs.

To be able to deal with dynamic Pareci context, each `Call` contains the context of the caller and whenever a `Call` is analysed by the worklist algorithm, it updates the analysis given this context. To be able to allow recursive and multiple sequential calls, the corresponding `Return` also updates the analysis given the context in the opposite direction to make sure the analysis is correct and does not infinitely grow. This is handled by the transfer function and is explained in detail for the liveness analysis in section 5.2.2.

We extend the `ProgramPoint` to support this by adding the Pareci context to the constructors of `Call` and `Return`.

```
data ProgramPoint = [...]
                  | Call   {ident :: Ident, pContext :: Context}
                  | Return {ident :: Ident, pContext :: Context}
                  | Exit   {ident :: Ident}
```

5.1.5. Monotone Framework instance

To conclude we can now formulate the Monotone Framework instance

$$\left\langle \widehat{\mathrm{L}}, \widehat{\mathcal{F}}, \mathrm{F}, \mathrm{E}, \iota, \widehat{\mathrm{f}_\iota} \right\rangle$$

for Pareci. Using $\widehat{\mathrm{L}} = \Delta \to \mathrm{L}$, where L is the relevant analysis lattice and Δ is the extra contextual information taken into account in the Monotone Framework instance. $\widehat{\mathcal{F}}$

= $\mathcal{P}((\Delta \to L) \to (\Delta \to L))$, F is the flow as defined in section 5.1.2, E are the extremal labels, which is the init program point in the case of a forward analyses and the final program points in backwards analyses, ι are the extremal values and therefore dependent on which analysis is done. $\widehat{f_l} : \mathcal{P}((\Delta \to L) -> (\Delta \to L))$ is as discussed in section 4.2.2, making use of $f_l : L \to L$.

For all analyses we can pick the same $\left\langle \Delta, F, E, \widehat{f_l} \right\rangle$ and we only have to specify the $\langle L, \iota, f_l \rangle$ which are dependent on the analysis chosen. For L we can get away by only specifying the $\bot$, $\sqcup$ and $\sqsubseteq$, which is in essence a bounded semi-lattice. The worklist algorithm uses only those operators. Some analyses define either bottom and join or top and meet, essentially the lattice flipped upside down. We stick to the notation using bottom and join. Top is defined as $\forall l \in L.l \sqsubseteq \top$ and $\forall l \in L.l \sqcup \top = \top$, essentially meaning that once you get to $\top$ you are "done".

5.2. Analyses on Pareci

Now that we have a Monotone Framework representation for Pareci, we can start doing program analyses on Pareci programs. Ultimately we want to do type analysis, but it is also interesting to look at different simpler analyses.

In the following sections we look at liveness analysis as an example of a widely used analysis type applied to Pareci in section 5.2.2. We devise two new analyses for Pareci. The first is a used fields analysis in section 5.2.3, which can help reduce the number of database calls. The second is a soft typing analysis in section 5.2.4, which warns the user of possible type conflicts. For each analysis we discuss possible input programs and desired analysis results.

5.2.1. Object Model

To be able to do a correct analysis and because of the fact that Pareci is a mainly used in data driven applications, we need to have a representation of our objects. We take our object model as described in section 2.3 as $M \subseteq \mathcal{P}(\mathrm{ObjectName} \times \mathrm{ObjectType})$.

Pareci object models mainly come from the data model specified in the `models.yml` file. This file can be read in by the analyser and are parsed to a `Class` object which basically has a list of properties with names and values. The values can be primitive types or names of other classes. In this case a class is analogous to a type. The Haskell definitions for this can be found in appendix D.1.

The object model classes can be extended by supplying more properties and types. We do this because more fields and methods can be added to the PHP-classes that correspond to the data objects in the `models.yml` file and thereby become available for Pareci.

We discuss types more in depth when we get to the type analysis in section 5.2.4.

5.2.2. Liveness analysis

Liveness analysis (LV) is a backwards analysis that can determine which variables are live. A variable is live at a program point if its value may be used in the future, otherwise it is dead. This information can for example be used to optimise a garbage collector by freeing the memory for objects that are no longer needed to finish the execution [Mul93].

We can also use live variable analysis here to find all variables used during the execution, but that are not defined.

In the case of a Pareci page, we create the Monotone Framework instance as described in section 5.1. To finish formalising the Monotone Framework instance we need to define $\langle \mathrm{L}, \iota, \mathrm{l_f} \rangle$ per section 5.1.5.

Live Variable Lattice

The analysis result is an element of our analysis lattice LVL, which is $\mathcal{P}(\mathrm{Var}^*)$, where Var^* is the set of variables in the analysed program and Var is as defined in section 5.1.3. We define the ordering $\sqsubseteq$ on this lattice as $\subseteq$, because more variables means less information. The bottom, the maximum amount of information, of the LVL is the empty set. As join ($\sqcup$) we must use set union ($\cup$).

Our extremal value ι defines the variables dat are requested as output of the whole application. For us this is the empty set, since we do not require any live variables upon termination.

With the Monotone Framework instance for this analysis it can be solved using the worklist algorithm. The undefined variables can then be read out at the `Start` node of each `Page` as a set of variables that were referenced on the page but never defined.

Transfer function

We define the transfer function in an incremental way, adding new important aspects as we introduce them.

The transfer function f can be formulated as below. The context ($\circ$) are transferred to the effect ($\bullet$) of the widget. The killed variables are subtracted from the analysis result, then the generated variables are added.

$$\begin{aligned} \mathrm{f}_{[w]^{\mathrm{l}}_{\mathrm{c}}} : & \quad \mathrm{LVL}_\circ \to \mathrm{LVL}_\bullet \\ \mathrm{f}_{[w]^{\mathrm{l}}_{\mathrm{c}}}(\mathit{analysis}) = & \quad \mathit{analysis} \setminus \mathrm{kill}\left([w]^{\mathrm{l}}_{\mathrm{c}}\right) \cup \mathit{gen}\left([w]^{\mathrm{l}}_{\mathrm{c}}\right) \end{aligned}$$

For each widget $[w]^{\mathrm{l}}_{\mathrm{c}}$ with label l and Pareci context c, we define gen and kill functions which define variables that are generated and killed on that node. The $\setminus$ and $\cup$ symbols represent respectively the set minus and union operators

We generate variables when they are used at a program point and kill variables that are defined. For each widget $\mathit{gen}\left([w]^{\mathrm{l}}_{\mathrm{c}}\right)$ contains all generated variables that occur in

one of their properties and $\mathrm{kill}\left([w]^{\mathrm{l}}_{\mathrm{c}}\right)$ contains all properties that they can produce. In both cases we also need to process any Pareci context information, which is discussed below in section 5.2.2. The kill and gen functions are:

$$\begin{aligned} \mathrm{kill}, \mathrm{gen}: &\quad \mathrm{ParsedWidget} \to \mathcal{P}(\mathrm{Var}) \\ \mathrm{gen}\left([w]^{\mathrm{l}}_{\mathrm{c}}\right) &= \bigcup \mathrm{vars}(\mathrm{c}, \mathrm{p}), \text{ where } \mathrm{p} \in \text{properties of } w, \text{ that are not child widgets} \\ \mathrm{kill}\left([w]^{\mathrm{l}}_{\mathrm{c}}\right) &= \mathrm{propNames}(w) \end{aligned}$$

where $\mathrm{vars}(\mathrm{p}, \mathrm{c})$ gives us the set of all variables used in p updated with Pareci context c. Since we filtered out the child widgets p is a `PropertyValue`, which can be a binding, expression or simple string. We ignore the string and return the binding or any bindings found in the expression. $\mathrm{propNames}(w)$ returns the names of widget w prepended with its `id`.

The flow also consists of `Call`, `Return`, `Exit`, `Start`, `End` and `EventLoop` program points. We also need to define the kill and gen functions for these points. In all cases we let them kill and generate the empty set, since they do not create or use variables. We consider the end of the program to be when the user session is ended. This can be triggered by a user action or happen after a time-out by not using the application. This way we can assume that no variable is live at the end of the program execution (and therefore ι is the empty set).

Killing Data Objects Some widgets kill some more variables than their direct properties, such as with `ObjectResource` and `Method`, we must know which properties objects have. This is captured in the object model, which is defined in section 5.2.1. A naïve approach is, whenever such an object comes into scope, to look it up in the objectmodel and add all possible fields on that object to the kill result. To deal with possible recursively defined objects and therefore an unlimited number of properties, we limit the maximum object depth[12] used by the kill function.

Killing scope Another special case here are scope generating widgets such as `Page`. All ID's defined somewhere within them and their children are available in that scope only. References to them on a page are also generated during the traversal of the flow. The scope generating widgets therefore also need to kill all ID's defined in that scope.

To deal with any scope changes we introduce the function $\mathrm{scopekill}$ that takes into account the scope stack s.

[12] Another, maybe better, but not implemented approach is to make the kill function dependent on the current analysis result, such that we only kill relevant elements. Since the current analysis result is finite, we can check this in finite time as well and be sure we looked deep enough in the object to find all relevant variables.

For a scoped widget $[w]^{l}_{c,s}$ with scope s, we define the following kill function:

$$\mathit{scopekill}\left([w]^{l}_{c,s}\right) = \mathrm{top}\,(s)$$
$$\mathit{scopekill}\,(w) = \emptyset$$

For scoped widgets the current scope (top of the scope stack) needs to be killed. For all other widget this kills nothing extra. The stack s can be precalculated before doing the analysis for non-action widgets and handled in the same way as context for action widgets and thus each element of s is a set of ID's defined on that level, stored as variables.

This results in an updated transfer function:

$$f_{[w]^{l}_{c,s}} : \quad \mathrm{LVL}_{\circ} \to \mathrm{LVL}_{\bullet}$$
$$f_{[w]^{l}_{c,s}}\,(\mathit{analysis}) = \quad \left(\mathit{analysis} \setminus \mathrm{kill}_{[w]^{l}_{c,s}} \cup \mathit{gen}_{[w]^{l}_{c,s}}\right) \setminus \mathit{scopekill}_{[w]^{l}_{c,s}}$$

The $\mathit{scopekill}_{[w]^{l}_{c,s}}$ set is removed last because gen also contains the ID and variables prefixed by the ID of the current scope, which we want killed as well.

Correct Pareci Context Another important factor is the Pareci context. Since some variables are only valid inside a context, which should stay correct. In the case of assignments inside a non-action widget, we already know the context in which a binding is used and the correct variables (in a scope) can be determined. For actions we must update the variables accordingly to any context changes.

Given a Pareci context c and LVL analysis result $\mathit{analysis}$, we can update the $\mathit{analysis}$ for widgets that are relative to the context. Binding variable set {`{a}`, `{b.c}`, `{#id}`} and context `{new}` can for example be updated to {`{new.a}`, `{new.b.c}`, `{#id}`}. Here the context is added for contextual variables (`a`, `b.c`) and ignored for non-contextual variables (`#id`). All contextual variables have a normal form that begins with `~_context` by which they can be recognised.

Since this analysis is backwards, all contextual variables in the analysis result need to be updated or reverted when respectively a `Call` or a `Return` is encountered. These context changes only occur when encountering `Call` or `Return`, since for all the other widgets the Pareci context can be determined statically. In a forward analysis the update and revert are done inversely. It should be the case that all variables in the analysis result between the call and return (during the action) are stripped from their Pareci context, since the whole action and everything in it is relative the call context. When leaving the action the context is again added to the variables because the action context is left.

The context update is handled by $A \triangleright_{w} c$. For all w where w is not an instance of `Call` or `Return` it is the identity. For `Call` or `Return` it updates the A as described below.

The graph in fig. 5.5 illustrates the contextual variable changes. This graph does not include (non-Pareci) analysis context for simplicity. The action generates two variables

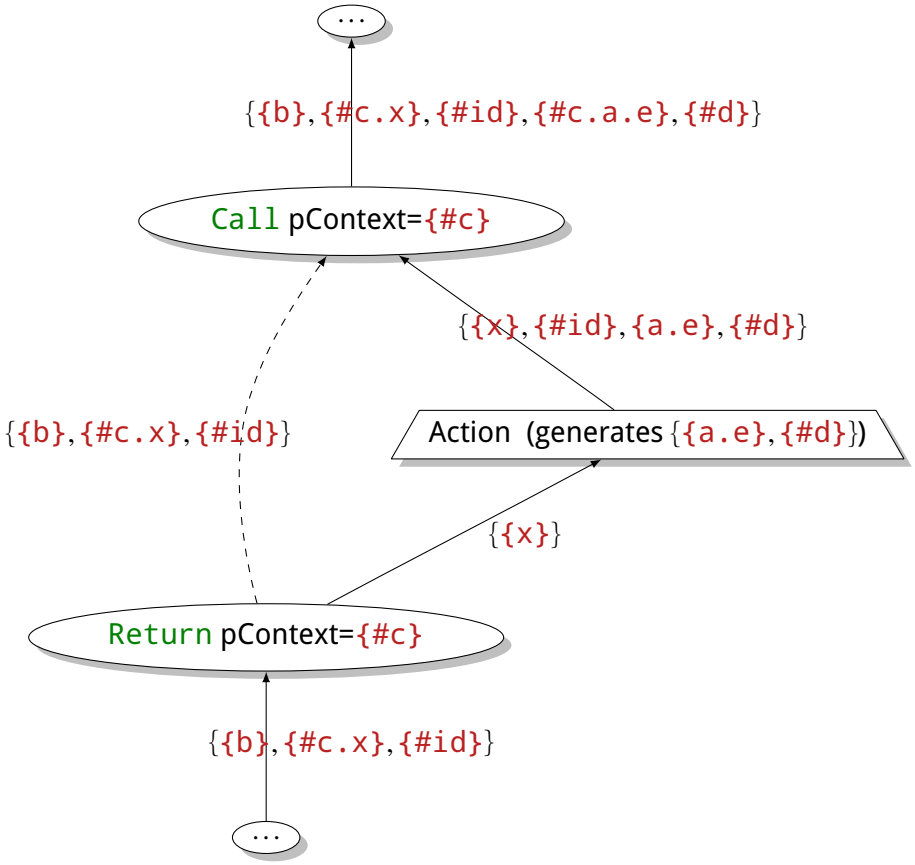


Figure 5.5.: Context update example

({{a.e},{#d}})). When we follow the flow from the bottom, we see that the Pareci context is removed from the analysis result while inside the action (the branch from Return to $Action$). The context is re-added when the action is finished (after the Call). The direct (dashed) branch from Return to Call is filled with a copy of the initial analysis result of Return for use in the binary interprocedural transfer function of Call.

More generally we have the pattern depicted in fig. 5.6, based on the interprocedural data flow analysis as discussed in section 4.2.2. Each node w is decorated with a context ($\circ$) and effect ($\bullet$) node. The Transfer function f_w is used to transfer the context to the effect. We make this distinction in this figure to be specific about the flow of analysis result values.

Starting at the bottom, we have an analysis context (non-Pareci) cs for which analysis result $g \in$ analysis lattice L is found. The node $[\texttt{Return}]^{l_r}_{\{c\}}$ has label l_r and associated Pareci context c. The edge $(\circ_{l_r}, \circ_{l_c})$ can be seen as redirecting the input analysis result of Return to Call. This edge is not needed in the flow, because the $f_{[\texttt{Call}]^{l_c}_{\{c\}}}$ can use the input of its corresponding $[\texttt{Return}]^{l_r}_{\{c\}}$ directly. This extra input for $f_{[\texttt{Call}]^{l_c}_{\{c\}}}$

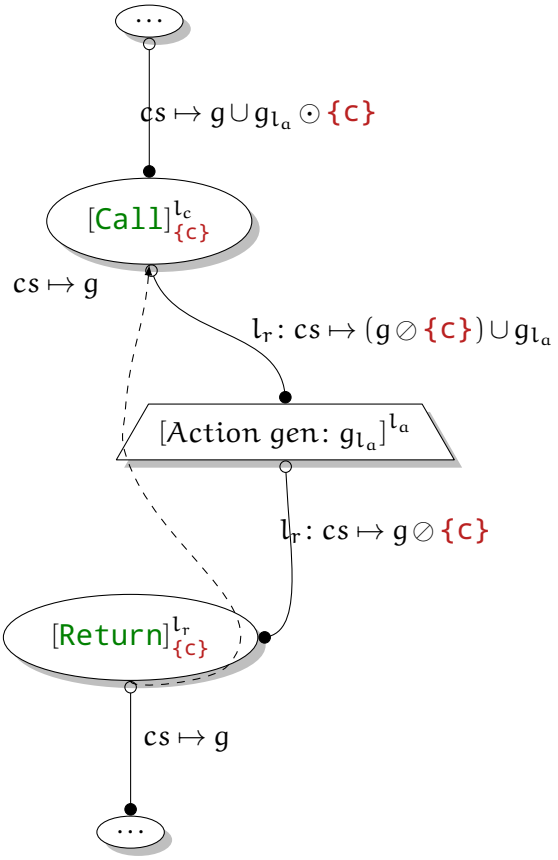


Figure 5.6.: Context update pattern

is handled by $f^{2A}_{l_c}(\{cs \mapsto g\})$; in our case this is the identity function, since we do not need to modify or update the input.

For the edge $(\bullet_{l_r}, \circ_{l_a})$, $f^{l}_{l_r}(\{cs \mapsto g\})$ is applied, which updates the analysis context by updating (with operator :) the calling context and subtracts ($\oslash$) the Pareci context c from the lattice value, resulting in reducing the values to those only relevant to the Pareci context corresponding with the function call. $Action$ generates the set g_{l_a} which should be added to the analysis value via the gen function, resulting in $(g \oslash \{c\}) \cup g_{l_a}$ for the effect of l_a.

The last step for this action call is combining the results in the call effect. The function handling `Call` is $f^{2}_{l_c} = f^{2A}_{l_r} \cup f^{2B}_{l_c}$. The result of edge $(\bullet_{l_a}, \circ_{l_c})$ is handled by $f^{2B}_{l_c}$; it updates (with $\odot$ applied to the correct context c) the result coming from the action effect. $f^{2}_{l_c}$ joins both results, resulting in $g \cup ((g \oslash \{c\}) \cup g_{l_a}) \odot \{c\}$.

We define $\oslash$ and $\odot$ to have a higher priority than $\cup$ to reduce the number of brackets. $A \oslash b$ subtracts the context b from every value $a \in A$, if it can not subtract b from a, a is omitted from the result, because a is not in context b thus $A \oslash b \subseteq A$. $A \odot b$ adds context b to every $a \in A$, thus $|A| \equiv |A \odot b|$. Therefore the derivation rule $A \cup (A \oslash B \odot B) \to A$ holds because subtracting context B and consequently updating it again with B results

in the same se.; Items removed by $\oslash$ stay in the collection because of $A\cup$. Now we can rewrite the result as follows:

$$
\begin{aligned}
& g \cup ((g \oslash \{c\}) && \cup g_{l_a}) && \odot \{c\} \\
\equiv & g \cup ((g \oslash \{c\} \odot \{c\}) && \cup g_{l_a} \odot \{c\}) \\
\equiv & g \cup g && \cup g_{l_a} \odot \{c\} \\
\equiv & g && \cup g_{l_a} \odot \{c\}
\end{aligned}
$$

This means that the original g is kept in the analysis result after handling an action and that the possible generated values g_{l_a} in context `c` are also added with respect to the correct Pareci context. This is exactly what we want.

The transfer function f for Live variable analysis becomes:

$$
\begin{aligned}
f_{[w]^l_{c,s}} : & \quad LVL_\circ \to LVL_\bullet \\
f_{[w]^l_{c,s}}(analysis) = & \quad \left(analysis \triangleright_w c \setminus kill_{[w]^l_{c,s}} \cup gen_{[w]^l_{c,s}}\right) \setminus scopekill_{[w]^l_{c,s}}
\end{aligned}
$$

Possible Pareci context changes are handled by $A \triangleright_w c$ which denotes the variables in A updated given widget w and context c. $\triangleright_w$ is the identity function for everything other than `Call` and `Return`. For `Call` and `Return` it subtracts and updates the Pareci context as described above.

Expected output given input

Simple Input:

- `Page` p containing
  - `ObjectResource` or with `id` `or`, `method` `getNew` and `object` `A`
  - `Var` v with `name` `v`, `value` `#or.value`
  - `Stack` s with `context` `v.C` with
    - `TextOutput` t `value` `x`
- ObjectModel M containing
  - Object A with fields $\{b, C, d\}$
  - Object C with fields $\{x, y\}$

  where capitals denote foreign fields/Classes and non-capitals are primitive (direct) fields.

Expected output:

- $\emptyset$

This means that there are no undeclared variables on p. There are two variables generated during this analysis. The variable `x` in t is generated using its context (which is inherited from its parent) `v.C.x`, which in turn gets more specified using the scope: `#or.value.C.x`. The other one is `#or.value` used by v. `#or.value.C.x` is killed by `or` using the information from M and `#or.value` gets killed because it is a property of `or`. The result is thus empty, which means the page is correct.

Missing fields Given the same input as above, we add another `TextOutput` t' to s with `value {z}`.

Expected Output:

- {`{#or.value.C.z}`}

This means that `#or.value.C.z` is used but never declared (because field z is not found in C in the object model).

Missing Var Input:

- `Page` p containing
  - `TextOutput` t `value {x}`

Expected Output:

- {`{x}`}

`{x}` is relative to the scope, but is not found, therefore it is reported as 'missing'. If a `Var` with `name x` was added, it would be correct.

Concluding

We have defined a liveness analysis that works for Pareci. It can be used to find out which variables are used, but never declared and we know that there is a program executions that possibly throws runtime errors when encountered.

Some examples of how the analysis should perform were defined in section 5.2.2.

We also discussed a way to take care of the Pareci context in a specific, and also more general way. This can now be used in the other analyses.

5.2.3. Used Fields analysis

Included and Excluded Fields

This is an analysis on `includeFields` and `excludeFields` to see which fields are always lazily fetched from the `ObjectResource` and which fields are always fetched but not used. It solves the problem sketched in section 1.1.2.

Both `includeFields` and `excludeFields` are a child widget of `ObjectResource` contain the fields that respectively should be selected in a query and ones that should not be selected. Only one should be used at any time. If both are specified `excludeFields` is ignored. When not specified, the default value for `includeFields` is $*$, which means that all fields on the `ObjectResource`'s `object` should be selected, hereby ignoring foreign fields. The default value for `excludeFields` is empty. If `excludeFields` has a value the selected fields are all fields from the `object`, minus the specified fields. The syntax for `include-` and `excludeFields` is a comma-separated list where the fields of the object can be specified, also fields of related objects can be specified by sequencing the objects split by a '.'. The symmbol $*$ can be used as a wildcard, which stands for all the fields on the object, except any foreign fields. It is similar to the `Id` type, except that a wildcard is allowed. Both properties are parsed using the `uu-parsinglib`. Let us call the parsed versions $\mathrm{includeFields}$ and $\mathrm{excludeFields}$, which both are subsets of Fields, where $\mathrm{Fields} = \mathcal{P}(\mathrm{Id})$. Any wildcards $*$ should be expanded to a list of primitive fields using the specified object model.

A side-note here is that `ObjectResource` is bound to a PHP object. This means that any specified also fields and methods are allowed in the `include-` and `excludeFields` that do not influence the number of queries. By default the fields that are extracted from the model are flagged as query relevant[13]. For the user defined classes it is a setting. This should be set if the methods use a field on the data object. This means that if all user defined classes correctly correspond to actual PHP classes and fields, this produces correct analysis results.

The analysis can look at the `includeFields` and `excludeFields` at the same time. The actual field list used in the query is basically the set of included fields minus the set of excluded fields. If one of them is not set then it has the default value as described above. Let us define fields as $\mathrm{includeFields} \setminus \mathrm{excludeFields}$. Then fields contains effectively the fields used by the database query that is run under the hood by Pareci.

It can be the case that fields of an object in fields, are not used on the page. This means that the content of those fields is always collected from the database, but never used. This is potential overhead and is reported by this analysis by returning $\mathrm{UnusedFields} \subseteq \mathrm{Fields}$. It can also be the case that fields are used on the page but not in fields. This means that they are collected lazily upon request, which results in an extra database query for every field collected in this way. The analysis also returns these with $\mathrm{LazyFields} \subseteq \mathrm{Fields}$.

Both the unused fields and the lazy field result can be determined per `ObjectResource` in the analysed program. The combined result UF of the analysis thus becomes $\mathcal{P}(\mathrm{ObjectResource}^* \times \mathcal{P}(\mathrm{LazyFields} \times \mathrm{UnusedFields}))$, where $\mathrm{ObjectResource}^*$ is the set of all `ObjectResource`s in the analysed program.

[13] In the current implementation this is not yet implemented. All the fields, both from the object models and the user defined classes are interpreted as query relevant.

Used Fields Lattice

To determine the values for $\mathrm{UnusedFields}$ and $\mathrm{LazyFields}$, we need to know which fields are used during a program execution. The same fields are also captured with the Liveness analysis. Since the result is different from Liveness analysis we can not just read out the result at the initial label of the analysis.

The lattice for the Used Fields analysis builts upon the Live Variable lattice by wrapping it. The type of the Used Fields Lattice (UFL) is an element $\mathcal{P}(\mathrm{LVL} \times \mathrm{UF})$. We collect the Liveness variables in the first position of the pair, and if applicable the results:w:$\mathrm{ObjectResource}$ in the second position. The ordering ($\sqsubseteq$) here is $\subseteq$ as well, because the more variables combined with object resources with found lazy and unused field the more information we have. $\sqcup$ is $\cup$ and bottom is the empty set. ι is the empty set.

This is a backwards analysis since it piggy backs on Liveness analysis. The backwards analysis makes sure we can output a suggested value for `includeFields` at the end of the analysis. A forward version is also possible where the `ObjectResource` adds the fields it creates to the analysis result and when used get removed again. At the end node of the analysis the remaining fields are labeled as unused, any extra used variables are lazily loaded (if available in the object model).

Transfer Function

We use the transfer function $\mathrm{f}^{\mathrm{LVL}}_{[w]^\iota}$ from the Liveness analysis the LVL for part of the lattice with the transfer function of LVL. In the case that the handled widget is a `ObjectResource`, it also adds the $\mathrm{LazyFields}$ and $\mathrm{UnusedFields}$ for that `ObjectResource` to the result.

$$
\begin{aligned}
&\mathrm{f}_{[w]^\iota} : \mathrm{UFL}_\circ \to \mathrm{UFL}_\bullet \\
&\mathrm{f}_{[w]^\iota}((\mathrm{ana}_{\mathrm{LVL}}, \mathrm{ana}_{\mathrm{UF}})) = \\
&\qquad \begin{cases} \left(\mathrm{f}^{\mathrm{LVL}}_{[w]^\iota}(\mathrm{ana}_{\mathrm{LVL}}), \mathrm{uf}_{[w]^\iota}(\mathrm{ana}_{\mathrm{LVL}}, \mathrm{ana}_{\mathrm{UF}})\right) & \text{if } w \text{ is an } \texttt{ObjectResource} \\ \left(\mathrm{f}^{\mathrm{LVL}}_{[w]^\iota}(\mathrm{ana}_{\mathrm{LVL}}), \mathrm{ana}_{\mathrm{UF}}\right) & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{uf}_{[w]^\iota} : \mathrm{LVL}_\circ \to \mathrm{UF}_\circ \to && \mathrm{UF}_\bullet \\
&\mathrm{uf}_{[w]^\iota}(\mathrm{ana}_{\mathrm{LVL}}, \mathrm{ana}_{\mathrm{UF}}) = && \text{let } \mathit{relevant} = \mathrm{ana}_{\mathrm{LVL}} \oslash w_{\mathrm{id}} \\
& && \text{in } \mathrm{ana}_{\mathrm{UF}} \sqcup (w, (\mathit{relevant} \setminus \mathrm{fields}_w, \mathrm{fields}_w \setminus \mathit{relevant}))
\end{aligned}
$$

If $\mathrm{f}_{[w]^\iota}$ does not encounter an `ObjectResource` it behaves just as the Liveness analysis and keeps the Used Fields part untouched. If it does encounter an `ObjectResource` w it using the Liveness results and any included or excluded fields to determine the lazy and unused fields w and adds it to the analysis. The value $\mathit{relevant}$ contains the fields relevant for the current `ObjectResource` by stripping the ID (w_{id})

of the resource. Value fields_w contains the fields used by the database query as described above in section 5.2.3 and is specific for w. The first position of the second pair denote the lazy fields, which are determined by subtracting the queried fields from the actual used relevant fields. The result for the unused fields is the inverse. The combined result is indexed by the `ObjectResource` w and added to the result.

In the initial program point we can read out the final result for each `ObjectResource` in the application. The extremal value ι for the Used Fields analysis is the empty set.

Expected output given input

Default fields Input:

- `Page` p containing
  - `ObjectResource` or with `id` equal to `or`, `method` with value `search` and `object A`.
  - and other widgets using (normalised) variables {`{#or.value.b}`, `{#or.value.C.x}`}
- ObjectModel M containing
  - Object A with fields $\{\mathrm{b}, \mathrm{C}, \mathrm{d}\}$
  - Object C with fields $\{\mathrm{x}, \mathrm{y}\}$

  where capitals denote foreign fields/Classes and non-capitals are primitive (direct) fields.

Desired output:

- $\mathrm{or} \mapsto (\mathrm{LazyFields} : \{\texttt{\{C.x\}}\}, \mathrm{UnusedFields} : \{\texttt{\{d\}}\})$

The external field `C.x` is queried at the moment it is requested. Since `includeFields` has its default value of `*`, it selects all fields of A, but `A.d` is not used on the page.

Include Fields Same input as above, but with an `includeFields` of `*, C.*`.

Desired output:

- $\mathrm{or} \mapsto (\mathrm{LazyFields} : \emptyset, \mathrm{UnusedFields} : \{\texttt{\{d\}}, \texttt{\{C.y\}}\})$

Now $\mathrm{LazyFields}$ is empty, because all required fields are already selected. Due to the nature of the wildcard, it now also selects another unused field `C.y`. This could be solved by specifying the exact fields used in the `includeFields`.

Exclude Fields Same input as Default fields, but with an `excludeFields` of `b,d`.
Desired output:

- $\mathrm{or} \mapsto (\mathrm{LazyFields} : \{\texttt{\{C.x\}}, \texttt{\{b\}}\}, \mathrm{UnusedFields} : \emptyset)$

Set $\mathrm{UnusedFields}$ is now empty because all fields are removed by the `excludeFields`. The $\mathrm{LazyFields}$ still fetches `C.x` lazily and now also the b aswell, because we excluded it for the initialisation query by specifying it in the `excludeFields`.

Concluding

This analysis detects when it is possible that too many fields are selected in the $\mathrm{UnusedField}$ result. It also detects when field are loaded lazily in the $\mathrm{LazyFields}$ result. Lazy fetched fields potentially blow up the number of database queries that are described in section 5.2.2. This analysis provides the Pareci programmer with information of whether this happens in the program. He or she can decide to act on this information.

This approach piggybacks on the Live variable analysis and depends on its transfer function to collect the variables used on the page. This means that the variables collected may be used in a program execution, but not necessarily in all program executions. It can be the case that the user of Pareci left some field lazily loaded on purpose, because in most cases the field is nog needed. It might proof fruitful to also have a variant of this analysis that reports the must-be-used variables, and thereby reporting the fields that are lazily fetched in all program executions. This can be achieved using the same lattice elements, but changing the join operator to intersection for the liveness part to make sure that only the variables used in all paths are collected.

5.2.4. Type analysis

Mostly similar to the analysis of LV, but now variables also carry type information, which have to be taken into account.

The idea is to do an approach similar to the Living Variables analysis, but forward in this case with types next to the variable names. We can use the same tranfer function but with the specifics swapped for `Call` and `Return`. We can learn the types by how variables are used or by looking at the object model which already contains the type information. When encountering Pareci scope or context changes we can still track the variable types although we can not directly map them to absolute variables. Analogous to the context handling using $\widehat{\mathrm{f_l}}$ in Liveness Analysis we can generate the same variables.

We take as lattice variables with associated types. Type Lattice (TL) is $\mathcal{P}(\mathrm{Var}^* \times \mathrm{Types_k})$. The variabel k is used to limit the type lattice as discussed below.

Let us first look at Types. The definition is found below.

$$\begin{aligned}
\mathit{Types} &::= \mathit{TSet} | \mathcal{P}(\mathit{TFun}) \\
\mathit{TSet} &::= \mathcal{P}(\mathit{TRep}) | \top \\
\mathit{TName} &::= \mathit{String} \\
\mathit{Primitive} &::= \mathit{Integer} | \mathit{Float} | \mathit{Decimal} | \mathit{Boolean} | \mathit{String} | \mathit{Collection}\ \mathit{TRep} \\
\mathit{TRep} &::= \mathit{Primitive} | \mathit{TName} \\
\mathit{TFun} &::= \mathit{TSet}^* \to \mathit{TSet}
\end{aligned}$$

TSet is a set of type representations TRep or $\top$ (Top), which can be everything. Having more TRep in a TSet means that the variable is less restricted to those types. The ordering of TSet is thus: $t_1 \sqsubseteq t_2 \sqsubseteq \ldots \sqsubseteq t_n \sqsubseteq \top$ where $t_i \in \mathit{TSet}$ for which $|t_i| \leqslant |t_j|$ for $i < j$ and $t_i \sqsubseteq \top$. $\top$ is thus used to bound the lattice.

We can again pick the ordering operator as subset equality ($\subseteq$) and the join as set union ($\cup$), since we want to decrease the knowledge of types that a variable can have. Both ι and $\bot$ are the empty set.

As before PHP functions and extra fields are handled by creating a custom class interface which can be read in addition to the object model. It can be used to both specify other PHP classes and methods, but also extend the classes of the object model.

Since the lattice Types is infinite, we use a widening operator (∇_k) as join to limit the depth. This makes sure our lattice is not infinite and therefore our analysis terminates. The type lattice Types_k using (∇_k) is bound by number k. This means that once the size of TSet becomes larger than k, the join converges to T. For $\forall t, t' \in \mathit{Types}, k \in \mathcal{N}$, we can define ∇_k as this:

$$t \nabla_k t' = \begin{cases} \top & \text{if } |t \cup t'| > k \\ t \cup t' & \mathit{otherwise} \end{cases}$$

where $|t|$ gives us either the size of the encapsulated TSet or $\mathcal{P}(\mathit{TFun})$.

We denote a variable v and its associated types t as $(v :: t)$. Given an Action call $[\texttt{Call}]^l_{\{c\}}$ to Action a which uses binding `{y}` as type t_y, we get as analysis result after the action call $(\{c.y\} :: t_y)$. Due to the Pareci context update it gets applied to the correct contextual value as illustrated in fig. 5.7.

At the end of the analysis we can check if every contraint matches the types as provided by the object model. Any discrepancies can then be reported to the user.

In addition to the Pareci context changes ($\triangleright_w$) as described in section 5.2.2 and which is handled by the transfer functions for `Call` and `Return`, widgets generate variables and types used and never remove them from the analysis.

The transfer function f for the Soft Typing analysis becomes:

$$\begin{aligned}
f_{[w]^l_{c,s}} &: & \mathit{TL}_\circ \to \mathit{TL}_\bullet \\
f_{[w]^l_{c,s}}(\mathit{analysis}) &= & \mathit{analysis} \triangleright_w c \cup \mathit{gen}_{[w]^l_{c,s}}
\end{aligned}$$

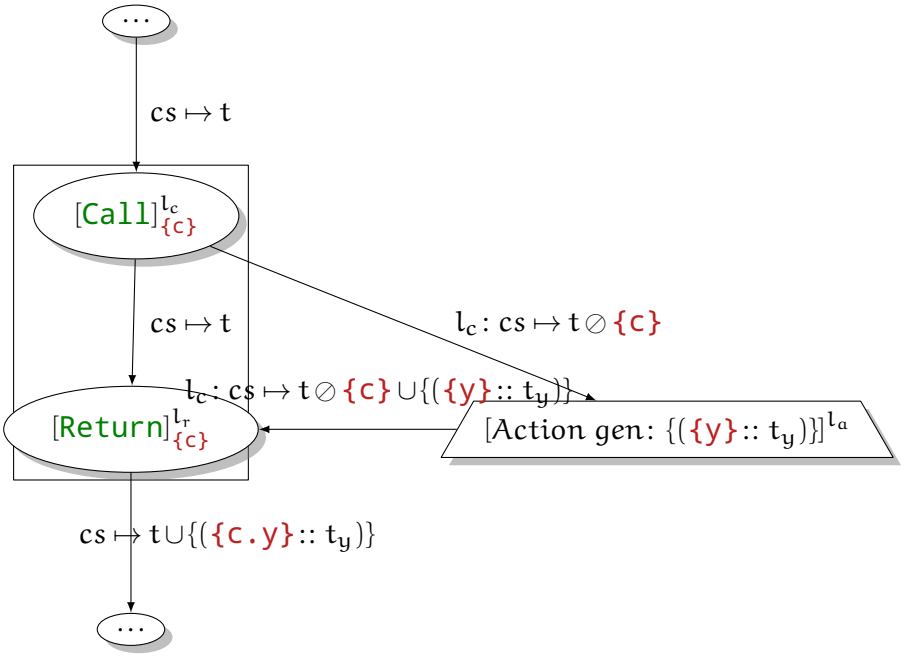


Figure 5.7.: Type analysis in an action call

Generated variables include the bindings used in the properties of the widgets and the corresponding types depend on how the bindings are used. All properties have a type associated with them, for example the `condition` property is required to be a boolean value, therefore any direct used bindings in the value of `condition` are added as a $\mathrm{Boolean}$. If a binding occurs inside an expression (starting with a $=$), its type depends on the operators used. Boolean operators such as AND, OR and XOR generate a $\mathrm{Boolean}$ type and numbers operators such as $+$, and $-$ generate Number types. Type inference on properties explains the inferencing and gathering of the variables and their derived types.

The widget definition (see section 3.1.2), also contain type information for each property of each widget.

Type inference on properties

To do correct type analysis at the widget level we have to first look at the property level. For each property we know which type it should have from the widget definition. Given property p with type τ and a property value v, we can determine the variables and present in v and their types. We define a function $\mathrm{pvVarsTypes} : \mathrm{PPropVal} \rightarrow \mathrm{TRep} \rightarrow \mathcal{P}(\mathrm{Var} \times \mathrm{Types})$, where the first argument is the property value v. The second the type as defined by the widget definition τ. Remember that $\mathrm{PPropVal}$ is a tree of either child widgets, bindings or expressions. We use an attribute grammar for passing down the requested type τ with a synthesised attribute and use an inherited attribute to bubble up the return value.

Consider the expression `={person.age} > 18` as value for a `condition` property. We can create the following derivation given that the type of `condition` is $\mathrm{Boolean}$.

$$\dfrac{\dfrac{}{=:: a \to a} \quad \dfrac{= \texttt{\{person.age\}} > 18 :: \mathrm{Boolean}}{\texttt{\{person.age\}} > 18 :: \mathrm{Boolean}}}{\texttt{\{person.age\}} :: \mathrm{Num} \qquad > :: \mathrm{Num} \to \mathrm{Num} \to \mathrm{Boolean} \qquad 18 :: \mathrm{Num}}$$

This natural deduction style of writing follows the tree structure of the expression data type. On top we have our expression with the requested type. Each line down is a next step in one of the branches. We know that $=$ keeps the type intact, therefore we split it in the first step. Because we know that $>$ is a function that takes two Num arguments we know that both `{person.age}` and 18 must be of type Num. The deriving is finished, since we have only axioms left. We collect the axioms and filter it such that we only have variables left, resulting (in this case) in $\{(\texttt{\{person.age\}} :: \mathrm{Num})\}$, which is the result of $\mathrm{pvVarsTypes}$.

Extra care has to be taken with the polymorphism turning up here. Num for example can be used for either $\mathrm{Integer}$ or Float. And the type of $=:: a \to a$ is even more polymorph, so we have deal with that as well.

In this example case the type of $>$ matches nicely with the requested type. In the case that we have a type mismatch, such as in the example `={person.age} > 'string'` between $>$ and `'string'`. We can still derive the type for `person.age`, because it does not change the type of $>$. The literal `'string'` has two types: Num requested by $>$ and String by looking at its value. We can issue a warning for the user that these types do not match.

Combining the results

At this moment all program points in the analysis have a corresponding set of variables and associated derived types. This results in a set, such as: $\{(\texttt{\{person\}} :: \mathrm{Person}), (\texttt{\{person.age\}} :: \mathrm{Num})\}$, for which it is not yet checked in any way that Person actually contains a field age of type Num. The $\sqsubseteq$ operator should check this and report a warning when incompatible types are found.

Concluding

We sketched a possible approach to type analysis. At the writing of this thesis the type analysis is not implemented, but given the foundation given by the other analyses it should not be too difficult. The lattice join and transfer function have to be specified, which can be adapted from the liveliness analysis implementation.

5.2.5. Results

In this section we discuss the output results of the prototype analyser tool. We run the analyser on the pages discussed in the "Expected output given input" sections of the Liveness Analysis and Used Fields Analysis.

Also we demonstrate the use of the analyser on a typical Pareci page. The source of the page and corresponding data model can be found in appendix E.1.1 and appendix E.1.2. An illustration of the rendered page can be seen in fig. 5.8. The top of the page contains a data entry form bound to a `Person` object via an `ObjectResource`. When the `LinkButton` with text `Save Person` is clicked an action is fired using the form's Pareci context, which calls the `save` method to store the new person in the database. Also the table view of all persons below is updated.

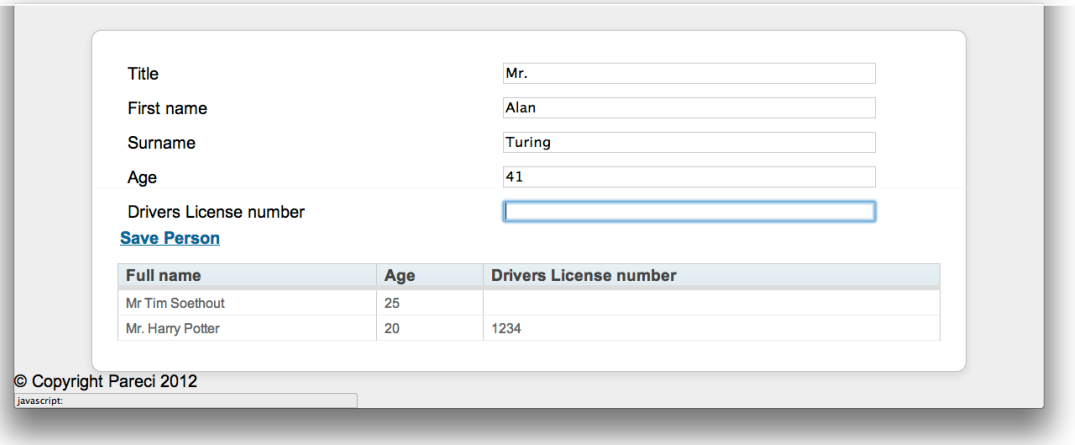


Figure 5.8.: Screenshot of page with typical Pareci

5.2.6. Liveness Analysis results

Expected output given input

We run the test cases as defined in the Expected output given input section of the Liveness analysis.

The input pages and data model files can be found in appendix E.2.

- The full output of the analyser run on the Simple test case:

```
================================
Analysing file: pages/simple.xml
Objectmodel file succesfully read in (yml).
XML successfully read in.
```

```
No errors/warnings. :)

Analysis results
----------------

Analysis Type: Contextual Live Variables Analysis
Fields/Variables used, but not defined (or not in Object Model):
Nothing
----------------
```

The result is “Nothing”, which is exactly what we expected.

- The results for the Missing fields test case are:

  ```
  - #or.value.C.z
  ```

  The result is the binding `#or.value.C.z`, which is exactly what we expected.

- The results for the the Missing Var test case are:

  ```
  - x
  ```

  The result is the binding x, which is exactly what we expected.

Results on a typical page

If we run the Liveness analysis on the typical page from appendix E.1, the result is:

```
Nothing
```

This means that the page is correct according to our analyser. To make the analysis more interesting we purposely add some mistakes to the page. The new page source can be found in appendix E.1.3. The following changes are made:

1. A non-existing action:

   ```
   <NumberInput label=”Age” value=”{age}” onchange=”{#check}”
   />
   ```

   We change `#checkAge` into `#check`

2. A misspelled/non-existing field:

   ```
   <TextInput label=”Surname” value=”{lastName}” />
   ```

   We change `surName` into `lastName`

3. A misspelled/non-existing field on a foreign object:

   ```
   <TextOutput value=”{DriversLicense.identification}” />
   ```

   We change `DriversLicense.number` into `DriversLicense.identification`

4. An non-existing field on a data object (in an Action):

   ```
   <a:Assign value="='Person saved to database: '.{name}" />
   ```

   We change `{firstName}.' '.{surName}` into `name`

5. Calling a method on a non-existing object (in an Action):

   ```
   <a:Method object="{person}" method="save" />
   ```

   We change `{}` into `person`

Now the output of the analysis is:

```
Errors:
Error: Could not find callable: "check"

Analysis results
----------------

Analysis Type: Contextual Live Variables Analysis
Fields/Variables used, but not defined (or not in Object Model):
- #allPersons.value.DriversLicense.identification
- #check
- #person.value.lastName
- #person.value.name
- #person.value.person
- #person.value.person.save
----------------
```

Let us discuss the results in the same order as the changes made:

1. The first thing that happens is the error which notes that a callable "check" can not be found, this is correct because there is not action defined with `id` equal to `check`. Since a call to `check` exists this is reported. The `#check` in the result is also present because the analysis looks for any used ID's on the flow.

2. Result `#person.value.lastName` is because the field `lastName` is non-existing.

3. The same holds for result `#allPersons.value.DriversLicense.identification`. The analysis also checks for existing fields on foreign objects.

4. Binding `name` is used inside the action and therefore relative to the Pareci context of the action which comes from the caller `LinkButton`. The analysis thus correctly handles Pareci context. This result is reported here because the field `name` does not exist within Person.

5. Results `#person.value.person` and `#person.value.person.save` both come from the last change. Field `person` is not found on the Pareci context of the call and therefore also its field (or in this case method) can not be found in the object model.

Concluding

It is thus the case that current analyser tool is able to support the detection of undefined and unresolvable bindings and Pareci context is handled correctly. The analyser using the Liveness analysis as defined in section 5.2.2 works as desired.

5.2.7. Used Fields Analysis results

Expected output given input

Again we run the test cases as defined in the Expected output given input section of the Liveness analysis.

The input pages and data model files can be found in appendix E.3.

- The full output of the analyser run on the Default fields test case:

```
Analysing file: pages/default-fields.xml
Objectmodel file succesfully read in (yml).
XML successfully read in.
No errors/warnings. :)

Analysis results
----------------

Analysis Type: Contextual Used Fields Analysis
Widget Widget_ObjectResource
object:-> "a"
method:-> "search"
id:-> "or"
   Lazy loaded fields: {C.x}
   Preloaded, but unused fields: {d}
----------------
```

  The result is exactly what we expected.

- The output of the Include Fields test case:

```
Lazy loaded fields:
Preloaded, but unused fields: {C.y,d}
```

The result is as expected.

- The output of the Exclude Fields test case:

```
Lazy loaded fields: {C.x,b}
Preloaded, but unused fields:
```

The result is as expected.

Results on typical page

If we run the Used Fields analysis on the typical page from appendix E.1, the result is:

```
Analysis Type: Contextual Used Fields Analysis
Widget Widget_ObjectResource
object:-> "Person"
method:-> "search"
id:-> "allPersons"
     Lazy loaded fields: {DriversLicense.number}
     Preloaded, but unused fields: {dad_id,dri_id,id,mom_id}
```

Field `DriversLicense.number` is lazily loaded, because by default only the direct (non-foreign) fields are prefetched. Also some other simple fields from Person are loaded, but that is not the performance issue.

After we add the following `includeFields`

```
<ObjectResource id="allPersons" object="Person"
                method="search">
  <Param name="includeFields"
         value="*, DriversLicense.number" />
</ObjectResource>
```

the result becomes:

```
Lazy loaded fields: {}
Preloaded, but unused fields: {dad_id,dri_id,id,mom_id}
```

Now the foreign field is also fetched with the first and only query for this request.

Concluding

Using the Used Fields analysis we can thus get some insight in which fields are always fetched lazily if they are used. We can also see which fields are always fetched from the database, but never used.

The implementation of the analysis works correctly and as specified in section 5.2.3.

6. Conclusions

We have looked at the programming framework Pareci. To improve the usability of Pareci for developers, we have used static program analysis in the form of data flow analysis to accomplish this.

Pareci is a Turing Complete language and able to do complex computations, although day-to-day use is mainly creating data-driven web applications. We have given a comprehensive explanation of how Pareci works in chapter 2. We created a parser and representation for Pareci Programs in chapter 3.

In chapter 4 we have briefly touched on how to do program analysis using data flow analysis and how we can use Monotone Frameworks to do this.

Next we created Monotone Framework instances for Pareci in chapter 5. The interesting part there was to make sure the program flow of the declaratively specified Pareci pages was correct and in the right order. Another interesting part is how to correctly handle Pareci context and scope, since they can not be statically determined, but are only known at runtime.

In the same chapter we also discuss analyses that can be done on Pareci that help solving the open problems sketched in the introduction in chapter 1.

The liveness analysis discussed in section 5.2.2 can be effectively used to detect where Pareci throws runtime exceptions. It detects where undefined references are used and reports this to the developer. Alongside of the trivial exceptions which are directly shown by Pareci on a first page request, it also detects exceptions which only occur after multiple actions triggered by a user of the Pareci program due to the way the analysis data flow control is modelled, and therefore helps for a developer using Pareci to finding bugs prematurely by reporting undefined variables. In that sense it is successful in reducing the number of runtime bugs. Of course it is still the developer's decision to act on the analysis results.

The used fields analysis discussed in section 5.2.3 returns for each `ObjectResource` on the page two sets of interesting fields on the `object` used for that `ObjectResource`. A set of lazily fetched database fields, which uses always an extra database query for each time they are required on the page and a set of unused fields which are always fetched from the database, but never used for displaying on the page. The developer using Pareci, can use these result sets to get a better insight in the page's number of database queries and decide to change value of the includeFields and excludeFields parameters to optimise the page.

An extra unimplemented analysis approach is the soft typing analysis described in section 5.2.4, which can be used to make sure that variables are sure to be used in the correct use cases. When implemented this analysis also points the developer to Pareci code that must be changed to decrease the number of possible runtime exceptions.

This all resulted in a generic implementation of data flow analysis using monotone frameworks and solving them using a worklist algorithm. The discussed analyses on Pareci except the type analysis are implemented using this framework. They are available as a command line tool that can be run on Pareci projects and individual pages. In appendix F a description of the tool and source code structure can be found.

Both implemented analyses are evaluated in section 5.2.5, where their expected in and output, as descibed in the respective analysis sections, is matched up with the implementation. A page with aspects of typical Pareci usage is also handled well by the implementation.

We have shown that it is possible to model the Pareci program data flow for use in data flow analysis. Using that flow we created Monotone Framework instances for a number of analyses for Pareci. The analyses results are useful for developers using Pareci to find runtime errors in their program code and get insight in the database queries needed for the page.

6.1. Future Work

6.1.1. Improvement of the analyses

Provided is an approach on how to implement data flow analysis on Pareci and a prototype implementation for it. It would be nice to improve the analysis using more advanced techniques, such as:

- Improving the parsing and representation:
  - At the moment the bidirectional parsing step that can be used to parse Pareci pages, recognizes more than Pareci strictly allows. The allowed types of widgets allowed as XML-children is larger than Pareci actually supports. The checking of the correct child widgets can be handled during a type analysis step, but it would be better to be able to handle this during the actual parsing step, since the allowed widget types are available. For this we can construct a smarter, type-dependent pickler to handle the translation between the XML-tree and the Pareci page representation.
  - The same holds for the Pareci representation in the implementation. At the moment we allow instances of all widgets as possible values for properties that should only contain specific widgets (following the Pareci specification). It would be better to be more strict, for example by making our representation of widget instances more strict by limiting the property value by making use of stricter data types within the Haskell type system. The first approach for representing widgets as discussed in section 3.1.1 tries to do such a thing by using phantom types, but is thereby stricter than we want. The more generalised approach as discussed in section 3.1.2 is far less restrictive and therefore easily models what we want, although not forcing enough strictness on the types of allowed widgets and property values by making

everything analysis time values for which we manually must check if they conform to our desired model. The ideal approach would be to have a hybrid approach which could give us the best of both worlds.

- There is no strict defined semantics for Pareci. chapter 2 sketches the intuition of how Pareci works. It would be nice to have a more formal specification of Pareci to be absolutely sure that the analyses are also correct.

- Improving the program flow:
  - Support the analysis of whole applications. At the moment we support a Pareci project by analysing each page in the project individually. In a real life application we have nested pages and values being passing to other pages. This can be added to the analyses by reading in and adding the pages that are statically declared to the flow. This works for most common situations. Pages can also be loaded dynamically using expressions and actions. We have two ways to deal with this. We can either assume worst case assume that every pages in the application can be used there, or let the user specify which pages are used on that location.

    Pages can be reached in three ways. It is (1) the starting page of the application, (2) loaded as content of a `Section` widget, or (3) triggered by a `Goto` widget. The first way is already supported. For the second and third ways, the target page can be added to the flow easily when it is known. For the third way some care has to be taken to make sure the context is handled correctly, since new pages do not inherit the context from their parent or calling page.
  - The current approach and implementation does not handle the possible program executions where delays come into play. In Pareci is possible to delay an action for a given amount of time. During waiting time other actions can also be triggered. This makes it possible to have interleaved action sequences sharing the state of the page. A partial solution here is to allow breaking and resuming of sequences of actions in the program flow where the delay feature is active. This would model many of the possibilities that the `delay` property offers.
  - By having a more concrete and formal semantics of the Pareci language, we can better argue and demonstrate that the approach chosen for the data flow representation of Pareci is correct. We can then also more easily see which features of Pareci are not supported yet in the analyses.
  - The current program flow representation is also not too strict on the Pareci semantics in all cases. It deals with context, scope and actions correctly, but can be made better by supporting specific branching of the control flow for handling conditions and permissions on widgets. A widget is only rendered or executed when its `condition` and `permission` properties are true. This could be easily modeled by taking inserting an extra well-known if-then-

else branch into the program flow, which would result in representing more possible program executions, but thereby also making the analysis more precise.

- For some events that can trigger actions, we know that certain events always occur before other events. An example situation is that an action triggered by the onkeydown is always called before the one triggered by onkeyup. Another example situation is the same event trigger is defined on both a widget and its children. Pareci specifies whether only one of the events is actually triggered or in which order both are triggered. The current approach used handles this by modelling all possible calling sequences in the program flow by using the *EventLoop* program point, but thereby specifying more possible program flows than Pareci actually allows. This is not a wrong approach, but it could be made more precise by allowing the program flow to only contain one of the possible event sequences. This can be done by introducing more *EventLoop* program points, each of them corresponding with an event trigger, in a sequential flow corresponding with the order of the handling of these events in Pareci. This chain of *EventLoop*s can then be connected to form a loop again and added instead of the current *EventLoop*.

- Building a Pareci compiler:

  - Often static analysis is used in compilers to optimise the generated code. Many of the parts needed to implement a compiler such as parsing the languages, interpreting some part of the code are already in place. These can be use to actually create an interpreter or compiler for Pareci in Haskell, making use of possible optimisations and the speed of a compiled language. With this we can bypass the whole PHP implementation and have an optimised, better system that can directly be used to host Pareci applications.
  - Another possibility is to try and use quasi- [qua12] or antiquoting [Hin11] for the binding and expression syntax, which effectively makes it an embedded domain specific language and makes it possible for the expressions to be compiled and interpreted by Haskell. This makes any parse errors occur at compile time in the form of type errors, which would result in fewer runtime errors. For this approach to work, the developer using Pareci must have the full Haskell development tools available for compiling the Pareci pages, because the files need to be compiled by the Haskell compiler.

- Improving the monotone framework implementation
  - The lattices and analysis context information should be specified more on a type level to allow for easier correctness of the analyses. For example a type for the call string might be `data CallString = CallString Num`

`[Ident]`, where Num specified the exact length of the call string on a type level. Much research is done in this area in the form of dependent type theory and already features supporting this are being implemented in the GHC compiler.

The same trick can be used for lattices where we now use runtime lattices, because our lattice elements (, for example $\mathbb{V}\mathrm{ar}^*$ is dependent on the actual variables present in the analysed program). These can hopefully be abstracted away on a type level in a clever way.

- The Turing completeness in section 2.2.2 proof makes use of a PHP object `ArrayList` to simulate the ticker tape. To argue a better case for sufficient computational complexity for Pareci in itself, without discussing its implementation, we would want to remove any use of specific PHP objects. It should be possible to emulate the ticker tape fully in Pareci only constructs as well by using nested pages to store the ticker tape to the left and right of the machine's head.
- Fully implement the Pareci type analysis as discussed in section 5.2.4 to have type analysis in Pareci. The type inference would make better use of the types given in the widget definitions, than the other specified analyses on Pareci.
- Improving the usability of the analyser tool
  - At the moment the analyses are implemented in a command line analyser tool that can be run on a Pareci project for which it runs the analyses on all pages found in that project and prints the results to the standard out. We can extend this by creating an extra output format that is machine readable. This can be used to display the analysis results in the editor used to write the pages, allowing it to run the analysis while writing the page. On Pareci pages of typical size the analysis runs in a matter of seconds, so this seems doable.

7. Bibliography

[BLS04] Arthur I Baars, Andres Löh, and S Doaitse Swierstra. Functional pearl parsing permutation phrases. *Journal of functional programming*, 14(06):635–646, 2004.

[Bra90] Lisa M Braz. Visual syntax diagrams for programming language statements. *ACM SIGDOC Asterisk Journal of Computer Documentation*, 14(4):23–27, 1990.

[CHH09] P. Camphuijsen, J. Hage, and S. Holdermans. Soft typing php. Technical report, Technical Report UU-CS-2009-004, Department of Information and Computing Sciences, Utrecht University, 2009.

[DS05] A. Dijkstra and S. Swierstra. Typing haskell with an attribute grammar. *Advanced Functional Programming*, pages 1–72, 2005.

[Fri11] Levin Fritz. Balancing cost and precision for python soft typing. Master's thesis, Utrecht University, October 2011.

[Hin11] R. Hinze. Typed quote/antiquote or: Compile-time parsing. *Journal of Functional Programming*, 21(03):219–234, 2011.

[hxt12] Haskell xml toolbox (hxt). http://www.haskell.org/haskellwiki/HXT, 2012.

[Jac05] G. Jackson. Securing perl with type inference. 2005.

[JMT09] S. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. *Static Analysis*, pages 238–255, 2009.

[Ken04] Andrew J Kennedy. Functional pearl pickler combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.

[KL94] David J King and John Launchbury. Lazy depth-first search and linear graph algorithms in haskell. *GLA*, pages 145–155, 1994.

[Mul93] Anne Mulkers. *Live data structures in logic programs, derivation by means of abstract interpretation*. Springer-Verlag, 1993.

[NNH04] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2004.

[PS11] Étienne Payet and Fausto Spoto. Static analysis of android programs. In *Automated Deduction–CADE-23*, pages 439–445. Springer, 2011.

[qua12] Quasiquotation. http://www.haskell.org/haskellwiki/Quasiquotation, 2012.

[RHP05] A. Rigo, M. Hudson, and S. Pedroni. Compiling dynamic language implementations, 2005.

[Sip06] M. Sipser. *Introduction to the Theory of Computation*, volume 27. Thomson Course Technology Boston, MA, 2006.

[SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[Soe11] Tim Soethout. Random access machines and quantum computing. *Paper for Models of Computation course*, July 2011.

[Sta07] Stefan Staiger. Static analysis of programs with graphical user interface. In *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, pages 252–264. IEEE, 2007.

[Swi09] S. Swierstra. Combinator parsing: A short tutorial. *Language Engineering and Rigorous Software Development*, pages 252–300, 2009.

[Thi05] P. Thiemann. Towards a type system for analyzing javascript programs. *Programming Languages and Systems*, pages 140–140, 2005.

[Tur36] Alan M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265, 1936.

[Wil12] Alexander Wilce. Quantum logic and probability theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*, volume Supplement, chapter The Basic Theory of Ordering Relations. Fall 2012 edition, 2012.

[xml08] Extensible markup language (xml) 1.0 (fifth edition). http://www.w3.org/TR/xml/, November 2008.

[yam13] Hackagedb: yaml. http://hackage.haskell.org/package/yaml/, 2013.

A. Turing Completeness programs

A.1. Simple machine counting 1's

This program is based on the same program used to proof the Turing Completeness of Random Access Machines I did for the course *Models of Computation* [Soe11]. The machine's head starts between two numbers encoded as sequential ones, after halting, it will have one number on the tape which is the sum of the two.

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action" onloadClient="{#run}" >

  <Page.resources>
    <ObjectResource id="arrayL" object="Util" method="getNewArrayList" >
      <Param value="=(1,1,1)"/>
    </ObjectResource>
    <ObjectResource id="arrayR" object="Util" method="getNewArrayList">
      <Param value="=(1,1,1,0,0)"/>
    </ObjectResource>

    <ObjectResource id="nextStepName" object="Util" method="getUnique" />
  </Page.resources>

  <Page.message>
    <Var name="tapeLeft" value="={#arrayL.value}" />
    <Var name="current" value="=0" />
    <Var name="tapeRight" value="={#arrayR.value}" />
    <Var name="state" value="=1" />
    <Var name="done" value="false" />
    <Var name="step" value="=1" />
  </Page.message>

  <Page.actions>
    <a:ActionList id="run">
      <!-- Start Machine configuration table -->
      <a:ActionList condition="=(NOT {done}) AND {state} == 1" >
        <a:ActionList condition="=(NOT {done}) AND {current} == 0" >
          <a:Log message="state 1, current 0" />
          <a:Assign field="{current}" value="=1" />
```

```
      <a:Trigger target="{#moveRight}" />
      <a:Assign field="{state}" value="=2" />
    </a:ActionList>
    <a:ActionList condition="=(NOT {done}) AND {current} == 1" >
       <a:Log message="state 1, current 1" />
      <a:Trigger target="{#moveRight}" />
      <a:Assign field="{state}" value="=1" />
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == 2" >
    <a:ActionList condition="=(NOT {done}) AND {current} == 0" >
       <a:Log message="state 2, current 0" />
      <a:Trigger target="{#moveLeft}" />
      <a:Assign field="{state}" value="=3" />
    </a:ActionList>
    <a:ActionList condition="=(NOT {done}) AND {current} == 1" >
       <a:Log message="state 2, current 1" />
      <a:Trigger target="{#moveRight}" />
      <a:Assign field="{state}" value="=2" />
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == 3" >
    <a:ActionList condition="=(NOT {done}) AND {current} == 0" >
       <a:Log message="state 3, current 0" />
      <a:Assign field="{state}" value="=0" />
    </a:ActionList>
    <a:ActionList condition="=(NOT {done}) AND {current} == 1" >
      <a:Log message="state 3, current 1" />
      <a:Assign field="{current}" value="=0" />
      <a:Assign field="{state}" value="=0" />
    </a:ActionList>
    <a:Assign field="{done}" value="true" />
  </a:ActionList>
  <!-- End Machine configuration table -->
  <a:ActionList id="nextStep" condition="={done}" >
    <a:Goto page="turingMachine" target="{#nextStepName.value}">
      <Var name="tapeLeft" value="{tapeLeft}" />
      <Var name="current" value="{current}" />
      <Var name="tapeRight" value="{tapeRight}" />
      <Var name="state" value="{state}" />
      <Var name="step" value="={step}+1" />
    </a:Goto>
  </a:ActionList>
</a:ActionList>
```

```
    <a:ActionList id="moveRight">
      <a:Method method="append" object="{tapeLeft}">
        <Param value="{current}" />
      </a:Method>
      <a:Method id="popRight" method="shift" object="{tapeRight}" />
      <a:Assign field="{current}" value="={#popRight.result}" />
      <a:Assign field="{done}" value="true" />
    </a:ActionList>
    <a:ActionList id="moveLeft">
      <a:Method method="unshift" object="{tapeRight}">
        <Param value="{current}" />
      </a:Method>
      <a:Method id="popLeft" method="pop" object="{tapeLeft}" />
      <a:Assign field="{current}" value="{#popLeft.result}" />
      <a:Assign field="{done}" value="true" />
    </a:ActionList>

  </Page.actions>

  <Page.content>
    <Stack>
      <TextOutput value="='step: ' . {step}" />
      <TextOutput value="='state: ' . {state}" />

      <Stack layoutMode="Horizontal" class="panel tmTape" >
        <TextOutput value="{tapeLeft}" />
        <TextOutput value="{current}" class="tmHead" />
        <TextOutput value="{tapeRight}" />
      </Stack>
      <Section name="={#nextStepName.value}" />
    </Stack>
  </Page.content>
</Page>
```

A.2. Turing Machine Simulation of Binary Addition

This Pareci simulated a Turing Machine that calculates the product of two binary encoded numbers. The head starts before the first number and the numbers are separated by symbol _. The program is slightly different than the first example in that it replaces the current page for each step. This is done because Pareci otherwise runs against the limit of recursive calls in PHP.

A small PHP script is written to generate the Pareci code that is the machine table

from the 5 tuples as specified in section 2.2.2.

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action" onloadClient="{#run}" >

  <Page.resources>
    <ObjectResource id="arrayL" object="Util" method="getNewArrayList" />
    <ObjectResource id="arrayR" object="Util" method="getNewArrayList">
      <Param value="=('1','0','1','1','0','_','1','0','1','0','1','1')"/>
    </ObjectResource>
  </Page.resources>

  <Page.message>
    <Var name="tapeLeft" value="={#arrayL.value}" />
    <Var name="current" value="='1'" />
    <Var name="tapeRight" value="={#arrayR.value}" />
    <Var name="state" value="='a'" />
    <Var name="done" value="=false" />
    <Var name="step" value="=1" />
  </Page.message>

  <Page.actions>
    <a:ActionList id="run">
      <!-- Begin Machine configuration table -->
      <a:ActionList condition="=(NOT {done}) AND {state} == 'a'" >
        <a:ActionList condition="=(NOT {done})AND ({current} == '_'OR {curren
          <a:Log message="state a, current _" />
          <a:Assign field="{current}" value="='_'" />
          <a:Trigger target="{#moveRight}" />
          <a:Assign field="{state}" value="='1'" />
          <a:Assign field="{done}" value="true" />
        </a:ActionList>
      </a:ActionList>
      <a:ActionList condition="=(NOT {done}) AND {state} == 'a'" >
        <a:ActionList condition="=(NOT {done})" >
          <a:Log message="state a, current *" />
          <a:Trigger target="{#moveRight}" />
          <a:Assign field="{state}" value="='a'" />
          <a:Assign field="{done}" value="true" />
        </a:ActionList>
      </a:ActionList>
      <a:ActionList condition="=(NOT {done}) AND {state} == '1'" >
        <a:ActionList condition="=(NOT {done})AND ({current} == '_'OR {curren
          <a:Log message="state 1, current _" />
```

```
        <a:Assign field="{current}" value="='_'" />
        <a:Trigger target="{#moveLeft}" />
        <a:Assign field="{state}" value="='2'" />
        <a:Assign field="{done}" value="true" />
      </a:ActionList>
    </a:ActionList>
    <a:ActionList condition="=(NOT {done}) AND {state} == '1'" >
      <a:ActionList condition="=(NOT {done})" >
        <a:Log message="state 1, current *" />
        <a:Trigger target="{#moveRight}" />
        <a:Assign field="{state}" value="='1'" />
        <a:Assign field="{done}" value="true" />
      </a:ActionList>
    </a:ActionList>
    <a:ActionList condition="=(NOT {done}) AND {state} == '2'" >
      <a:ActionList condition="=(NOT {done})AND ({current} == '0')" >
        <a:Log message="state 2, current 0" />
        <a:Assign field="{current}" value="='_'" />
        <a:Trigger target="{#moveLeft}" />
        <a:Assign field="{state}" value="='3x'" />
        <a:Assign field="{done}" value="true" />
      </a:ActionList>
    </a:ActionList>
    <a:ActionList condition="=(NOT {done}) AND {state} == '2'" >
      <a:ActionList condition="=(NOT {done})AND ({current} == '1')" >
        <a:Log message="state 2, current 1" />
        <a:Assign field="{current}" value="='_'" />
        <a:Trigger target="{#moveLeft}" />
        <a:Assign field="{state}" value="='3y'" />
        <a:Assign field="{done}" value="true" />
      </a:ActionList>
    </a:ActionList>
    <a:ActionList condition="=(NOT {done}) AND {state} == '2'" >
      <a:ActionList condition="=(NOT {done})AND ({current} == '_'OR {curren
        <a:Log message="state 2, current _" />
        <a:Assign field="{current}" value="='_'" />
        <a:Trigger target="{#moveLeft}" />
        <a:Assign field="{state}" value="='7'" />
        <a:Assign field="{done}" value="true" />
      </a:ActionList>
    </a:ActionList>
    <a:ActionList condition="=(NOT {done}) AND {state} == '3x'" >
      <a:ActionList condition="=(NOT {done})AND ({current} == '_'OR {curren
        <a:Log message="state 3x, current _" />
```

```
    <a:Assign field="{current}" value="='_'" />
    <a:Trigger target="{#moveLeft}" />
    <a:Assign field="{state}" value="='4x'" />
    <a:Assign field="{done}" value="true" />
  </a:ActionList>
</a:ActionList>
<a:ActionList condition="=(NOT {done}) AND {state} == '3x'" >
  <a:ActionList condition="=(NOT {done})" >
    <a:Log message="state 3x, current *" />
    <a:Trigger target="{#moveLeft}" />
    <a:Assign field="{state}" value="='3x'" />
    <a:Assign field="{done}" value="true" />
  </a:ActionList>
</a:ActionList>
<a:ActionList condition="=(NOT {done}) AND {state} == '3y'" >
  <a:ActionList condition="=(NOT {done})AND ({current} == '_'OR {curren
    <a:Log message="state 3y, current _" />
    <a:Assign field="{current}" value="='_'" />
    <a:Trigger target="{#moveLeft}" />
    <a:Assign field="{state}" value="='4y'" />
    <a:Assign field="{done}" value="true" />
  </a:ActionList>
</a:ActionList>
<a:ActionList condition="=(NOT {done}) AND {state} == '3y'" >
  <a:ActionList condition="=(NOT {done})" >
    <a:Log message="state 3y, current *" />
    <a:Trigger target="{#moveLeft}" />
    <a:Assign field="{state}" value="='3y'" />
    <a:Assign field="{done}" value="true" />
  </a:ActionList>
</a:ActionList>
<a:ActionList condition="=(NOT {done}) AND {state} == '4x'" >
  <a:ActionList condition="=(NOT {done})AND ({current} == '0')" >
    <a:Log message="state 4x, current 0" />
    <a:Assign field="{current}" value="='x'" />
    <a:Trigger target="{#moveRight}" />
    <a:Assign field="{state}" value="='a'" />
    <a:Assign field="{done}" value="true" />
  </a:ActionList>
</a:ActionList>
<a:ActionList condition="=(NOT {done}) AND {state} == '4x'" >
  <a:ActionList condition="=(NOT {done})AND ({current} == '1')" >
    <a:Log message="state 4x, current 1" />
    <a:Assign field="{current}" value="='y'" />
```

```
      <a:Trigger target="{#moveRight}" />
      <a:Assign field="{state}" value="='a'" />
      <a:Assign field="{done}" value="true" />
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == '4x'" >
    <a:ActionList condition="=(NOT {done})AND ({current} == '_'OR {curren
      <a:Log message="state 4x, current _" />
      <a:Assign field="{current}" value="='x'" />
      <a:Trigger target="{#moveRight}" />
      <a:Assign field="{state}" value="='a'" />
      <a:Assign field="{done}" value="true" />
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == '4x'" >
    <a:ActionList condition="=(NOT {done})" >
      <a:Log message="state 4x, current *" />
      <a:Trigger target="{#moveLeft}" />
      <a:Assign field="{state}" value="='4x'" />
      <a:Assign field="{done}" value="true" />
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == '4y'" >
    <a:ActionList condition="=(NOT {done})AND ({current} == '0')" >
      <a:Log message="state 4y, current 0" />
      <a:Assign field="{current}" value="='1'" />
      <a:Assign field="{state}" value="='5'" />
      <a:Assign field="{done}" value="true" />
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == '4y'" >
    <a:ActionList condition="=(NOT {done})AND ({current} == '1')" >
      <a:Log message="state 4y, current 1" />
      <a:Assign field="{current}" value="='0'" />
      <a:Trigger target="{#moveLeft}" />
      <a:Assign field="{state}" value="='4y'" />
      <a:Assign field="{done}" value="true" />
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == '4y'" >
    <a:ActionList condition="=(NOT {done})AND ({current} == '_'OR {curren
      <a:Log message="state 4y, current _" />
      <a:Assign field="{current}" value="='1'" />
      <a:Assign field="{state}" value="='5'" />
```

```
    <a:Assign field="{done}" value="true" />
  </a:ActionList>
</a:ActionList>
<a:ActionList condition="=(NOT {done}) AND {state} == '4y'" >
  <a:ActionList condition="=(NOT {done})" >
    <a:Log message="state 4y, current *" />
    <a:Trigger target="{#moveLeft}" />
    <a:Assign field="{state}" value="='4y'" />
    <a:Assign field="{done}" value="true" />
  </a:ActionList>
</a:ActionList>
<a:ActionList condition="=(NOT {done}) AND {state} == '5'" >
  <a:ActionList condition="=(NOT {done})AND ({current} == 'x')" >
    <a:Log message="state 5, current x" />
    <a:Assign field="{current}" value="='x'" />
    <a:Trigger target="{#moveLeft}" />
    <a:Assign field="{state}" value="='6'" />
    <a:Assign field="{done}" value="true" />
  </a:ActionList>
</a:ActionList>
<a:ActionList condition="=(NOT {done}) AND {state} == '5'" >
  <a:ActionList condition="=(NOT {done})AND ({current} == 'y')" >
    <a:Log message="state 5, current y" />
    <a:Assign field="{current}" value="='y'" />
    <a:Trigger target="{#moveLeft}" />
    <a:Assign field="{state}" value="='6'" />
    <a:Assign field="{done}" value="true" />
  </a:ActionList>
</a:ActionList>
<a:ActionList condition="=(NOT {done}) AND {state} == '5'" >
  <a:ActionList condition="=(NOT {done})AND ({current} == '_'OR {curren
    <a:Log message="state 5, current _" />
    <a:Assign field="{current}" value="='_'" />
    <a:Trigger target="{#moveLeft}" />
    <a:Assign field="{state}" value="='6'" />
    <a:Assign field="{done}" value="true" />
  </a:ActionList>
</a:ActionList>
<a:ActionList condition="=(NOT {done}) AND {state} == '5'" >
  <a:ActionList condition="=(NOT {done})" >
    <a:Log message="state 5, current *" />
    <a:Trigger target="{#moveRight}" />
    <a:Assign field="{state}" value="='5'" />
    <a:Assign field="{done}" value="true" />
```

```
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == '6'" >
    <a:ActionList condition="=(NOT {done})AND ({current} == '0')" >
      <a:Log message="state 6, current 0" />
      <a:Assign field="{current}" value="='x'" />
      <a:Trigger target="{#moveRight}" />
      <a:Assign field="{state}" value="='a'" />
      <a:Assign field="{done}" value="true" />
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == '6'" >
    <a:ActionList condition="=(NOT {done})AND ({current} == '1')" >
      <a:Log message="state 6, current 1" />
      <a:Assign field="{current}" value="='y'" />
      <a:Trigger target="{#moveRight}" />
      <a:Assign field="{state}" value="='a'" />
      <a:Assign field="{done}" value="true" />
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == '7'" >
    <a:ActionList condition="=(NOT {done})AND ({current} == 'x')" >
      <a:Log message="state 7, current x" />
      <a:Assign field="{current}" value="='0'" />
      <a:Trigger target="{#moveLeft}" />
      <a:Assign field="{state}" value="='7'" />
      <a:Assign field="{done}" value="true" />
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == '7'" >
    <a:ActionList condition="=(NOT {done})AND ({current} == 'y')" >
      <a:Log message="state 7, current y" />
      <a:Assign field="{current}" value="='1'" />
      <a:Trigger target="{#moveLeft}" />
      <a:Assign field="{state}" value="='7'" />
      <a:Assign field="{done}" value="true" />
    </a:ActionList>
  </a:ActionList>
  <a:ActionList condition="=(NOT {done}) AND {state} == '7'" >
    <a:ActionList condition="=(NOT {done})AND ({current} == '_'OR {curren
      <a:Log message="state 7, current _" />
      <a:Assign field="{current}" value="='_'" />
      <a:Trigger target="{#moveRight}" />
      <a:Assign field="{state}" value="='0'" />
```

```
        <a:Assign field="{done}" value="true" />
      </a:ActionList>
    </a:ActionList>
    <a:ActionList condition="=(NOT {done}) AND {state} == '7'" >
      <a:ActionList condition="=(NOT {done})" >
        <a:Log message="state 7, current *" />
        <a:Trigger target="{#moveLeft}" />
        <a:Assign field="{state}" value="='7'" />
        <a:Assign field="{done}" value="true" />
      </a:ActionList>
    </a:ActionList>
    <!-- End Machine configuration table -->

    <a:ActionList id="nextStep" condition="={done}" >
      <a:Goto page="turingMachineMult">
        <Var name="tapeLeft" value="{tapeLeft}" />
        <Var name="current" value="{current}" />
        <Var name="tapeRight" value="{tapeRight}" />
        <Var name="state" value="{state}" />
        <Var name="step" value="={step}+1" />
      </a:Goto>
    </a:ActionList>
  </a:ActionList>

  <!-- Ticker tape modification -->
  <a:ActionList id="moveRight">
    <a:Log message="Move Right" />
    <a:Method method="append" object="{tapeLeft}">
      <Param value="{current}" />
    </a:Method>
    <a:Method id="popRight" method="shift" object="{tapeRight}" />
    <a:Assign field="{current}" value="={#popRight.result}" />
  </a:ActionList>
  <a:ActionList id="moveLeft">
    <a:Log message="Move Left" />
    <a:Method method="unshift" object="{tapeRight}">
      <Param value="{current}" />
    </a:Method>
    <a:Method id="popLeft" method="pop" object="{tapeLeft}" />
    <a:Assign field="{current}" value="{#popLeft.result}" />
  </a:ActionList>
</Page.actions>

<Page.content>
```

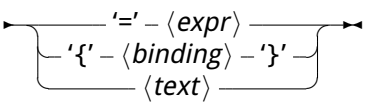
```
    <Stack id="content">
      <TextOutput value="='step: ' . {step}" />
      <TextOutput value="='state: ' . {state}" />
      <Stack layoutMode="Horizontal" class="panel tmTape" >
        <TextOutput value="{tapeLeft}" />
        <TextOutput value="{current}" class="tmHead" />
        <TextOutput value="{tapeRight}" />
      </Stack>
    </Stack>
  </Page.content>
</Page>
```

B. Property Value Grammar and Data Types

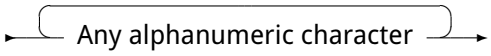
The grammar for the expression and binding syntax is displayed below in the form of railroad diagrams [Bra90] for easy reading. The entry point for the syntax is ⟨*expr*⟩. It is read from left to right and a non-terminal (between < and >) points to the definition also found below and defined in the same way.

We start with a parser that encapsulates property values using the binding and expressions parsers defined below.

⟨*propertyValue*⟩ ::=

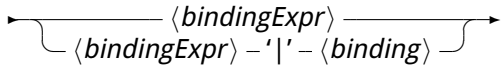


⟨*text*⟩ ::=

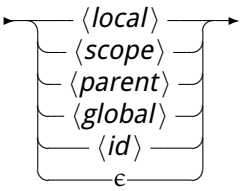


The binding syntax (without braces).

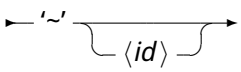
⟨*binding*⟩ ::=



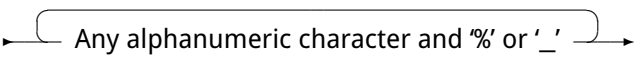
⟨*bindingExpr*⟩ ::=



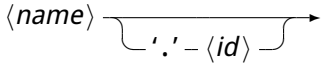
⟨*local*⟩ ::=



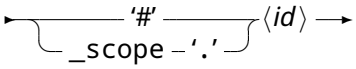
⟨*name*⟩ ::=

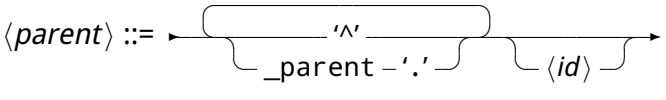


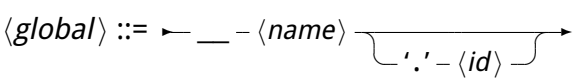
⟨*id*⟩ ::=



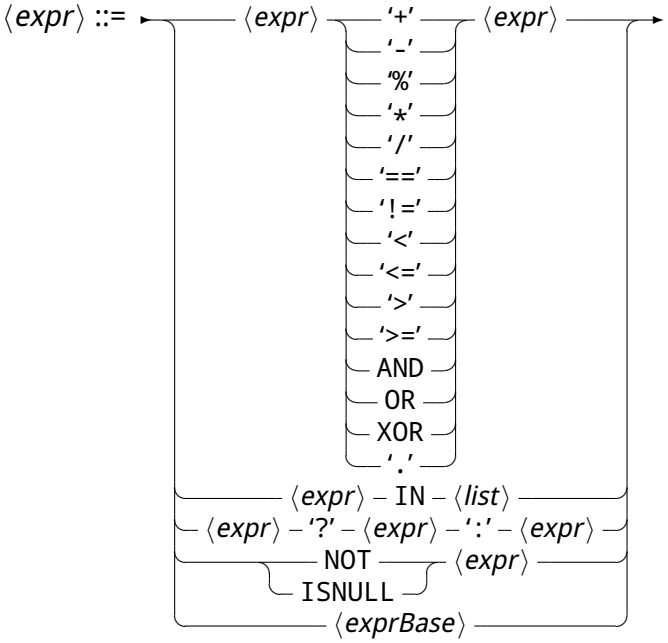
⟨*scope*⟩ ::=

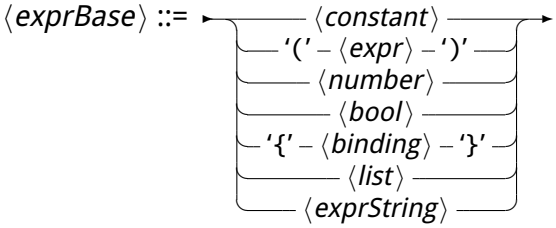


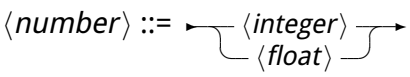


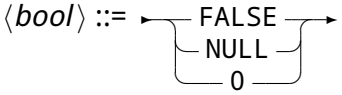


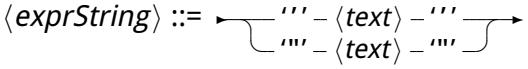
The syntax for the expression.

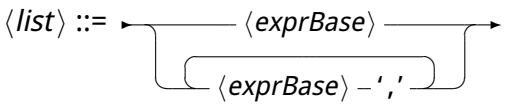


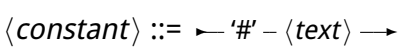












All parts are captured in the `PropertyValue`, `Expr` and `Binding` data types in Haskell in the `Language.Pareci.Properties.PropertyValue` module.

```
data PropertyValue = Nil
                   | String String
                   | Weak Expr
                   | Strong Binding
data Expr = Number     Number
          | Binding    Binding
          | Text       String
          | Constant   String
          | Bool       Bool
          | BiOp       BiOp Expr Expr
          | UnaryOp    UnaryOp Expr
          | List       [Expr]
          | In         Expr Expr
          | IfThenElse Expr Expr Expr

data BiOp = Eq
          | Neq
          | Lt
          | Lte
          | Gt
          | Gte
          | Concat
          | Plus
          | Min
          | Mult
          | Div
          | Mod
          | And
          | Or
          | Xor

data Number = Integer Int
            | Float Double
```

```
data UnaryOp = IsNull
             | Not

type Name = String

data Id = Id [Name]

data Binding = BLocal Id
             | BContext Id
             | BGlobal Name Id
             | BParent Binding
             | BScope Id
             | BOr Binding Binding
```

C. Maximal Fixed Point

C.1. Worklist algorithm

Implementation of the worklist algorithm to compute the Maximum Fixed Point as discussed in section 4.2.1 in algorithm 1.

This implementation makes use of the `State` monad to be able to abstract over the imperative while loop.

```
worklist :: (JoinSemiLattice lat, PartialOrd lat, Monotone expr
            , Show (Analysed expr), Show lat, DotLatex (Analysed expr)
            , DotLatex lat, DotLatex (Analysis expr lat), Groupable expr
            , (Grouped expr ~ Analysed expr))
            => AnalysisInstance lat expr -> FlowOut (Analysed expr) lat
worklist inst = (MFP analysis, MFP $ M.mapWithKey f analysis)
  where
    m = expr inst
    dir = direction inst
    bot = bottom inst
    i = initial inst
    f = transfer inst
    -- initialisation
    (anaFlow, e) = case dir of
      Forward  -> (reverse $ postOrder (flow m) (init m), [init m])
      Backward -> (map swap $ postOrder (flow m) (init m), final m)
    w = nub anaFlow -- inital worklist same as flow
    initAnalysis = fromList $ map select (labels m)
      where
        select lab =  ( lab
                      , if lab `elem` e
                        then i
                        else bot -- bottom from BoundedLattice
                      )
    analysis = fst $ execState ( do while (not . null . snd)
                                      $ do (analysis, wl) <- get
                                           (step anaFlow f)
                                    gets fst -- return analysis from state
                               ) (initAnalysis, w)
```

```
-- | Does one worklist algorithm iteration given transfer and join function
step :: (JoinSemiLattice lat, PartialOrd lat, Eq expr, Ord expr
        , Show lat, Show expr, DotLatex expr, DotLatex lat
        , DotLatex (Analysis expr lat))
        => Flow expr -> Transfer expr lat -> MonotoneState expr lat ()
step fl f =
  do (analysis, wl) <- get
     if null wl then return ()
       else let ((l, l') : restWl) = wl
                fanalysisl         = f l (analysis ! l)
                analysisl'         = analysis ! l'
                (newAnalysis, newWorklist)
                  = if not $ fanalysisl `leq` analysisl'
                    then
                      (insert l' ( analysisl' `join` fanalysisl) analysis
                      , [ (l', l'')  | (t,l'') <-  fl , t == l' ] ++ restWl
                      )
                    else (analysis, restWl)
            in  put (newAnalysis, nub newWorklist)

-- | Utility function representing a while loop in the State Monad
while :: (s -> Bool) -> State s () -> State s ()
while test body =
  do st <- get
     if (test st)
       then do modify (execState body)
               while test body
       else return ()

-- | calculating post order traversal of graph `fs` given
--   starting node `cur`
postOrder fs cur = snd $ helper fs cur []
  where helper :: Eq a => [(a,a)] -> a -> [a] -> ([a], [(a,a)])
        helper fs cur visited =
          let result = [ f | f@(l,l') <- fs, l == cur]
              visited' = cur : visited
              unvisitedChildren = [ l' | f@(l,l') <- fs, l == cur
                                       , l' `notElem` visited']
              op c (vis, res) = let (vis', res') = helper fs c vis
                                in (vis', res ++ res')
              (visitedChildren, resultChildren) =
                foldr op (visited', []) unvisitedChildren
          in (visitedChildren, resultChildren ++ result)
```

C.2. Embellished Worklist algorithm

Implementation of the worklist algorithm to compute the Maximum Fixed Point as discussed in section 4.2.2 in algorithm 2 for an Embellished Monotone FrameWork instance.

```
-- | Worklist algorithm
worklist :: (JoinSemiLattice lat, PartialOrd lat, EmbellishedMonotone expr
            , Show (Analysed expr), Show lat, DotLatex (Analysed expr)
            , DotLatex lat, DotLatex (Analysis expr lat), Groupable expr
            , (Grouped expr ~ Analysed expr))
            => AnalysisInstance lat expr -> FlowOut (Analysed expr) lat
worklist inst = (MFP analysis, MFP $ M.mapWithKey f analysis)
  where
    m = expr inst
    dir = direction inst
    bot = bottom inst
    i = initial inst
    f = transfer inst
    f2 = interTransfer inst
    -- initialisation
    postorderFlow = postOrder (flow m) (init m)
    (anaFlow, fl,  e, inter) = case dir of
      Forward  ->
        (reverse postorderFlow, flow m, [init m], interFlow m)
      Backward ->
        (map swap postorderFlow, map swap (flow m), final m
        , map (\(lc,ln,lx,lr) -> (lr, lx, ln, lc)) (interFlow m))
    -- inital worklist same as flow,
    -- add possible left out (unreachable..) nodes from postorder
    w = nub (anaFlow ++ (fl \\ anaFlow))
    initAnalysis = fromList $ map select (labels m)
      where
        select lab =  ( lab
                      , if lab `elem` e
                        then i
                        else bot -- bottom from BoundedLattice
                      )
    analysis = fst $ execState ( do while (not . null . snd)
                                      $ do (analysis, wl) <- get
                                           (step w inter f f2)
                                    gets fst -- return analysis from state
                               ) (initAnalysis, w)
```

```
-- | Does one worklist algorithm iteration given transfer and join function
step :: (JoinSemiLattice lat, PartialOrd lat, Eq expr, Ord expr, Show lat
        , Show expr, DotLatex expr, DotLatex lat, DotLatex (Analysis expr lat)
        => Flow expr -> InterFlow expr -> Transfer expr lat
        -> InterTransfer expr lat -> MonotoneState expr lat ()
step fl inter f f2 =
  do (analysis, wl) <- get
     if null wl then return ()
       else let ((l, l') : restWl) = wl
                mInter = find (\(lc,ln,lx,lr) -> l == lr) inter
                fanalysisl          = case mInter of
                  Nothing
                    -> f l (analysis ! l) -- default transfer function
                  Just (lc,ln,lx,lr)
                    -> f2 l (f l $ analysis ! lc) (analysis ! l)
                analysisl'          = analysis ! l'
                (newAnalysis, newWorklist)
                  = if not $ fanalysisl `leq` analysisl'
                    then
                      (insert l' ( analysisl' `join` fanalysisl) analysis
                      , [ (l', l'')  | (t,l'') <-  fl , t == l' ] ++ restWl
                      )
                    else (analysis, restWl)
            in  put (newAnalysis, nub newWorklist)
```

D. Object Model

D.1. Haskell class and type representation

```
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE TemplateHaskell, TypeSynonymInstances, FlexibleInstances #-}
module Language.Pareci.ObjectModel.Types where

import Safe (readMay)
import Data.Yaml (decode, encode, FromJSON, ToJSON, decodeFile)
import Data.List (groupBy, sort)
import Data.Aeson.TH
import Data.ByteString.Char8 (pack)
import Control.Monad (liftM)
import Debug.Trace

data Primitive a = Integer | Float | Decimal
                 | Boolean | String | Collection a
                 deriving (Show, Eq, Ord, Read)
type TName = String

-- | Set of types
data TSet = TSet [TRep]
          | Any
          deriving (Show, Eq, Ord, Read)

data TRep = Primitive (Primitive TRep) -- ^ Represents primitive types
          | Name TName
          deriving (Show, Eq, Ord, Read)

data TFun a = [a] :-> a
          deriving (Show, Eq, Ord, Read)

-- | Types used in analysis
data Types = Types     TSet -- ^
           | Functions [TFun TSet]
           deriving (Show, Eq, Ord, Read)

-- | Object model
```

```haskell
-- | Object model to denote
type ObjectModel = [Object]

data Object = ADT TName [(TName, TIdent)]
              -- ^ An ADT represents a custom data type and objects fields
            deriving (Show, Eq, Ord, Read)

data TIdent = TPrim (Primitive TIdent)
            | TName TName -- reference to object
            | TFun  (TFun TIdent)
            | Enum [String]
            | TAny -- ^ unparsable/unknown types
            | Self -- ^ special case where return type is the current type
            | Void -- ^ Special for functions
            deriving (Show, Eq, Ord, Read)

emptyObjectModel :: ObjectModel
emptyObjectModel = []

exampleObjectModel :: ObjectModel
exampleObjectModel = [ ADT "Person" [ ("id", TPrim Integer)
                                    , ("name", TPrim String)
                                    , ("Avatar", TName "Picture")
                                    , ("getNew", TName "Person")
                                    ]
                     , ADT "Picture" [ ("url", TPrim String) ]
                     ]

select :: ObjectModel -> TName -> [Object]
select om name = filter (\(ADT n _) -> n == name) om


-- | Default/ parent methods that are added to objects/ADTS
type ClassName = TName
type ParentClass = TName
type FieldName = TName
data Class = Class ClassName
                   (Maybe ParentClass)
                   [(FieldName, TIdent)]
           deriving (Show, Eq, Ord, Read)
type Classes = [Class]

$( liftM concat $ mapM (deriveJSON id) [''TRep, ''TSet, ''Primitive
```

```
                                        , ''TIdent, ''TFun, ''Class] )

classesToOM :: Classes -> ObjectModel
classesToOM classes = map adt classes
  where
    adt (Class cn pc fields) = ADT cn $ map (changeInFields cn) fields
                               ++ fieldsOf cn pc
    fieldsOf :: ClassName -> (Maybe ParentClass) -> [(FieldName, TIdent)]
    fieldsOf _ Nothing = []
    fieldsOf cn (Just c) =
      let parents = filter (\(Class n _ _) -> c == n) classes
      in concat [ map (changeInFields cn) fields ++ fieldsOf cn pn
                | (Class _ pn fields) <- parents ]

-- | instantiate self to actual Ident
changeSelfToName :: TName -> TIdent -> TIdent
changeSelfToName name Self = TName name
changeSelfToName name (TPrim (Collection ident)) =
  TPrim $ Collection $ changeSelfToName name ident
changeSelfToName name (TFun (sets :-> set)) =
  TFun $ (map (changeSelfToName name) sets) :-> changeSelfToName name set
changeSelfToName _ ident = ident
changeInFields name (n, field) = (n, changeSelfToName name field)

mergeClasses :: Classes -> Classes -> Classes
mergeClasses left right =
  let combine = left ++ right
      dups = groupBy (\(Class ln lpn lfields) (Class rn rpn rfields)
                      -> ln == rn)
                     (sort combine)
            -- ^ sort because groupBy only detects sequential duplicates
      union classes = foldr1 op classes
        where
          op l@(Class ln lpn lfields) r@(Class rn rpn rfields) =
            if lpn /= rpn
            then error $
                 "2 Classes defined with same name, but different parent: \n"
                 ++ show l ++ " and \n" ++ show r
            else Class ln lpn (lfields ++ rfields)
  in map union dups
```

E. Test pages

E.1. A Typical Pareci Page

Below is the source code of a typical Pareci page containing a data entry form and a table displaying a database data.

E.1.1. Page

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action">
  <Page.resources>
    <ObjectResource id="person" object="Person" method="getNew" />
    <ObjectResource id="allPersons" object="Person" method="search" />
  </Page.resources>
  <Page.actions>
    <a:ActionList id="saveAction">
      <a:Method object="{}" method="save" />
      <a:Assign field="{#output.value}"
          value="='Person saved to database: '.{firstName}.' '.{surName}" />
      <a:Refresh target="{#allPersons}" />
      <a:Refresh target="{#all}" />
    </a:ActionList>
    <a:Refresh id="checkAge" target="{#form}" />
  </Page.actions>
  <Page.content>
    <Stack id="all">
      <FormLayout id="form" context="{#person.value}">
        <TextInput label="Title" value="{title}" />
        <TextInput label="First name" value="{firstName}" />
        <TextInput label="Surname" value="{surName}" />
        <NumberInput label="Age" value="{age}" onchange="{#checkAge}" />
        <TextInput condition="={age} >= 18" id="dln"
                   label="Drivers License number"
                   value="{DriversLicense.number}" />
        <LinkButton text="Save Person" onclick="{#saveAction}" />
        <TextOutput id="output" />
      </FormLayout>
      <Table items="{#allPersons.value}">
```

```
        <TextOutput label="Full name"
                    value="={title}.' '.{firstName}.' '.{surName}" />
        <NumberOutput label="Age" value="{age}" />
        <TextOutput label="Drivers License number"
                    value="{DriversLicense.number}" />
      </Table>
    </Stack>
  </Page.content>
</Page>
```

E.1.2. Data Model

```
Person:
  tableName: tblperson
  columns:
    id:
      name: id
      primary: true
      type: integer(11)
      autoincrement: true
    title: string(10)
    firstName: string(100)
    surName: string(100)
    age: integer(3)
    dri_id: integer(11)
    dad_id: integer(11)
    mom_id: integer(11)
  relations:
    DriversLicense:
      local: dri_id
      foreign: id
    Mom:
      local: mom_id
      foreign: id
      class: Person
    Dad:
      local: dad_id
      foreign: id
      class: Person

DriversLicense:
  tableName: tblDriversLicense
  columns:
    id:
```

```
      name: id
      primary: true
      type: integer(11)
      autoincrement: true
    number: string(10)
    photo: blob()
```

E.1.3. A Typical Pareci Page with added mistakes

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action">
  <Page.resources>
    <ObjectResource id="person" object="Person" method="getNew" />
    <ObjectResource id="allPersons" object="Person" method="search" />
  </Page.resources>
  <Page.actions>
    <a:ActionList id="saveAction">
      <a:Method object="{person}" method="save" />
      <a:Assign field="{#output.value}"
                value="='Person saved to database: '.{name}" />
      <a:Refresh target="{#allPersons}" />
      <a:Refresh target="{#all}" />
    </a:ActionList>
    <a:Refresh id="checkAge" target="{#form}" />
  </Page.actions>
  <Page.content>
    <Stack id="all">
      <FormLayout id="form" context="{#person.value}">
        <TextInput label="Title" value="{title}" />
        <TextInput label="First name" value="{firstName}" />
        <TextInput label="Surname" value="{lastName}" />
        <NumberInput label="Age" value="{age}" onchange="{#check}" />
        <TextInput condition="={age} >= 18" id="dln"
                   label="Drivers License number"
                   value="{DriversLicense.number}" />
        <LinkButton text="Save Person" onclick="{#saveAction}" />
        <TextOutput id="output" />
      </FormLayout>
      <Table items="{#allPersons.value}">
        <TextOutput label="Full name"
                    value="={title}.' '.{firstName}.' '.{surName}" />
        <NumberOutput label="Age" value="{age}" />
        <TextOutput label="Drivers License number"
                    value="{DriversLicense.identification}" />
      </Table>
```

```
    </Stack>
  </Page.content>
</Page>
```

E.2. Liveness Analysis test pages

E.2.1. Simple

Page

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action">
  <Page.message>
    <Var name="v" value="{#or.value}" />
  </Page.message>
  <Page.resources>
    <ObjectResource id="or" object="A" method="getNew" />
  </Page.resources>
  <Page.content>
    <Stack context="{v.C}">
      <TextOutput value="{x}" />
    </Stack>
  </Page.content>
</Page>
```

Data Model

```
A:
  columns:
    b: string
    d: string
  relations:
    C: C

C:
  columns:
    x: string
    y: string
```

E.2.2. Missing Fields

Page

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action">
  <Page.message>
```

```
    <Var name="v" value="{#or.value}" />
  </Page.message>
  <Page.resources>
    <ObjectResource id="or" object="A" method="getNew" />
  </Page.resources>
  <Page.content>
    <Stack context="{v.C}">
      <TextOutput value="{x}" />
      <TextOutput value="{z}" />
    </Stack>
  </Page.content>
</Page>
```

Data Model

The same model file as for Simple is used.

E.2.3. Missing Var

Page

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action">
  <Page.content>
    <TextOutput value="{x}" />
  </Page.content>
</Page>
```

Data Model

The object model is empty.

E.3. Used Fields Analysis test pages

E.3.1. Default fields

Page

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action">
  <Page.resources>
    <ObjectResource id="or" method="search" object="A" />
  </Page.resources>
  <Page.content>
    <Stack>
      <TextOutput value="{#or.value.b}" />
```

```
        <TextOutput value="{#or.value.C.x}" />
    </Stack>
  </Page.content>
</Page>
```

Data Model

The same model file as for Simple is used.

E.3.2. Include fields

Page

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action">
  <Page.resources>
    <ObjectResource id="or" method="search" object="A">
      <Param name="includeFields" value="*, C.*" />
    </ObjectResource>
  </Page.resources>
  <Page.content>
    <Stack>
      <TextOutput value="{#or.value.b}" />
       <TextOutput value="{#or.value.C.x}" />
    </Stack>
  </Page.content>
</Page>
```

E.3.3. Exclude fields

Page

```
<Page xmlns="urn:Widget" xmlns:a="urn:Widget_Action">
  <Page.resources>
    <ObjectResource id="or" method="search" object="A">
      <Param name="excludeFields" value="b, d" />
    </ObjectResource>
  </Page.resources>
  <Page.content>
    <Stack>
      <TextOutput value="{#or.value.b}" />
      <TextOutput value="{#or.value.C.x}" />
    </Stack>
  </Page.content>
</Page>
```

F. Implementation

The implementation of the approach as described are property of Eljakim IT, but are available for the graders. This appendix describes the outline of the Haskell modules (and submodules).

F.1. Main

The main `analyser` program is typically run from a Pareci application directory with a pages folder and a `models.yml` with the data objects definitions.

Al the executables can me build by using the `Makefile` or by using `cabal`. For this the latest (svn) version of `uuagc` is needed. With the $-v$ flag analyser also outputs a pdf with a rendered Graphviz representation of the flow with corresponding analysis values. The `dot` executable from Graphviz is required in the path for this.

F.2. Pareci representation

- Widget representation: `Language.Pareci.Types` and `Language.Pareci.WidgetDefinition`
- Types and object model representation: `Language.Pareci.ObjectModel`
- Property values, Expressions and binding syntax: `Language.Pareci.Properties` and parsers: `Language.Pareci.Properties.*Parser`
- Pickler for Pareci for translating from XML-tree to the widget representation: `Language.Pareci.Pickle`

F.3. Monotone Framework

- Monotone framework representation: `Language.Pareci.Analysis.Montone`
- Worklist algorithm implementation: `Language.Pareci.Analysis.Worklist`
- General analysis types: `Language.Pareci.Analysis.AnalysisTypes`

This part is separate from rest of analysis. The worklist algorithm takes an instance of `AnalysisInstance` which parameters are an analysis lattice (`JoinSemiLattice lat, PartialOrd lat`) and a expression on which the necessary Monotone Framework instance properties (`Monotone expr`) are available. If the analysis is specified as an `AnalysisInstance`, we can run the analysis

- Monotone framework instance for Pareci:
  `Language.Pareci.Analysis.Flow`

The instance for Pareci is defined as an attribute grammar and converted to corresponding haskell code using uuagc.

F.4. Analyses on Pareci

`Language.Pareci.Analysis` contains two backwards analyses that run on Pareci.

- `Language.Pareci.Analysis.LiveVariablesAnalysis` for detecting if all used variables are declared
- `Language.Pareci.Analysis.UsedFieldsAnalysis` for detecting surplus database queries