Extending an Eclipse-Plugin for Curry by Features for Program Analysis, Type-Checking and Debugging

Lennart Spitzner

Master's Thesis
submitted in March 2014

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe für Programmiersprachen und Übersetzerkonstruktion

Advised by: Prof. Dr. Michael Hanus

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

Abstract

This work is a continuation of the work of Marian Palkus on an Integrated Development Environment (IDE) for the declarative programming language *Curry*.

The existing IDE was extended by adding features that provide the Curry programmer with more and better direct feedback about the validity of his or her source-code. This includes the capabilities to annotate compile-time errors in the source-code and to detect and display a large portion of type errors without even invoking the Curry interpreter. The Curry Analysis Server System (CASS) is a flexible tool for analyzing declarative programs. Access to this tool from the IDE had already been planned and partially implemented; This work advances on that and provides a functional, albeit simple, user interface for CASS in the IDE.

Another focus was the integration of debugging facilities into the IDE. There exist several approaches to debugging declarative languages with *needed narrowing* such as Curry. For the IDE, we implement observational debugging, where the user can choose expressions to observe at run-time. To allow future extensions of this functionality, a generic interface for expression debuggers was added.

Contents

Chapter 1

Introduction

1.1 Motivation

Multiple criteria must be met to make a programming language successful. The most obvious one is of course the language itself: It must be consistent, must not be too complex and it must not be too far from familiar grounds as to be understandable to new programmers. Yet it also needs to add something new and worthwhile in comparison to existing languages, for otherwise the effort of switching to a new language would be too large.

The declarative programming language Curry is an effort to combine two grand classes of programming languages:

▷ functional programming languages, represented by Haskell, providing a high-level style of programming and a strong static type system that enforces that all functions are pure, i.e. free of side-effects.

▷ logical programming languages, represented by Prolog, featuring a declarative approach, where a program consists of a series of logical rules, and the execution is a query containing free variables. The language then searches for possible assignments of these variables that can be deduced from the rules.

Another criterion for the success of a language is the amount of existing code in the language and of features already implemented. Does the language have a standard library containing common functionality, and frameworks that provide generic solutions to specific types of problems? If such basics were to be missing, the workload of the application programmer increases significantly. Also, there is the risk that multiple implementations of such basics are produced instead, non of which feature-complete or properly maintained.

But what is the meaning of "success" for a language? The main metric would be usage: How much code is being produced in the language. So the ease of producing new code is detrimental for the success, and this is what this work belongs to: the tooling support that enables the programmer to write, analyze and debug code.

Integrated Development Environments (IDEs) are one main part of modern software development. Compared to a simple text editor they add programming (and programming language) specific functionality to provide centralized access to the tools provided with the programming language and to give the programmer fast feedback about the static validity of his code. For example, one basic feature of and IDE is syntax checking of source-code without invoking an interpreter or compiler. This way, the programmer can quickly detect a large part of possible compilation errors.

1.1.1 Curry

Curry is a general purpose functional logic programming language, i.e. it aims to combine both the functional and the logical programming paradigms. It is based (especially for the functional aspects) on the programming language Haskell: Both Curry's syntax and its type system are only slight modifications of Haskell's. Curry is strongly statically typed and features *type inference* to automatically derive types. From logic programming, Curry inherits non-deterministic operations, constraint programming and logical variables. The evaluation strategy employed by Curry is *needed narrowing* that corresponds to Haskell's *lazy evaluation* extended to the logical aspects of Curry. Curry is standardized in the *Curry Report* [Mic12] (sometimes shortened to just "report").

We will describe the language and its ecosystem in more detail in Section 2.1.

1.1.2 Basis: Thesis by Marian Palkus

This work is based on the work of Marian Palkus, who produced a plugin for the Eclipse IDE as his diploma thesis [Pal12]. This plugin, shortly named *Curry IDE*, uses the language framework *Xtext* to add Curry-specific features to the basic Eclipse IDE. These features include syntax highlighting, syntax checking and the ability to define library settings. Xtext also supports a certain class of semantic checking: cross-referencing of the objects in the source code; this was implemented as well.

We will use the phrase "Palkus IDE" to refer to our predecessor's final version of the IDE, in contrast to any features that were added to the IDE as part of this thesis.

1.2 Similar Software

Emacs is a general purpose extensible editor that can be used for program development. For this editor there exists an extension (called *mode*) that provides Curry-specific features, most notably syntax highlighting and the ability to search for the declaration of any objects in the current module. It is also possible to open a system shell in Emacs and run the compiler in that; hence after making edits, the programmer can, with a few keystrokes, switch to the console and recompile the program.

For Haskell, many different IDEs exist. For several editors (Vim, Emacs, KDevelop) there are extensions providing language-specific syntax highlighting; KDevelop also features certain project management options. *Leksah* is an IDE specifically for Haskell (and written in Haskell) that incorporates several more advanced features including debugging support, continuous recompilation, and type checking for sections of code.

There also exists an Eclipse plugin for Haskell: *EclipseFP* supports syntax and error/warning highlighting, import management, test support and the integration of several external tools (compiler, debugger, static style checker).

1.3 Goals

In general terms, the goal of this thesis is to improve the functionality of the Curry IDE, both by fixing/finishing the existing setup and by adding certain new features. The external static program analysis tool *CASS* had been integrated into the Palkus IDE, but this work had not been completed because *CASS* still had been in development at the time. So a first task was to complete this work and make *CASS* accessible from within the IDE.

A second goal is to add better feedback to the Curry programmer about the validity of the source-code. Currently, the IDE checks and displays only the syntactic validity. This can be improved by taking advantage of Curry's static type system: Type errors should be detected and annotated in the IDE. The IDE is not a complete Curry compiler and certain semantic compile-time errors will not be detected.

Hence, in addition to type-checking, the IDE should be able to parse the output of the Curry compiler and annotate the errors in the editor.

Finally, we would like to include debugging functionality in the IDE. We will describe possible approaches to debugging in functional logic languages in the next chapter. Our goal is to add support for *observational debugging* to the IDE, i.e. the ability to trace the evaluation (and optionally, the resulting value) of specific expressions in the program at run-time.

Chapter 2

Foundations and Technologies

2.1 Curry

2.1.1 Basic Structure of Curry Programs

The structure and syntax of Curry programs is for the most part equivalent to Haskell's. A program consists of one more *modules*. Each module is defined in a separate file and has an identifier that corresponds to its position in the file system, e.g. a module *Data.List* would be defined in the file "Data/List.curry". Thus, modules are structured as a tree. The modules can *import* other modules in order to make their elements accessible. As its core, a module contains a list with[1]:

▻ *type synonyms* such as
`type String = [Char] - read:  a List of Characters is called "String"`

▻ *data declarations*:

```
data Tree a = Leaf a         -- a Tree over a type a is either a leaf,
            | Node (Tree a) a (Tree a)         -- or a binary node
                                               -- with two sub-nodes.
```

▻ *function declarations*:
`plusOne = (+) 1`

The definition of a function consists of an optional type-signature plus one or more *equations*. We could, for example, add a type signature to the function above:

```
-- type-signature:
plusOne :: Int -> Int    -- plusOne takes an integer as parameter
                         -- and returns an integer.
```

[1]technically, this enumeration misses fixity declarations which were omitted for brevity.

```
-- exactly one equation (in this case):
plusOne = (+) 1          -- We partially apply the (+) operator.
                         -- (+) :: Int -> Int -> Int
                         -- That is, providing the first parameter yields
                         -- the remaining type Int -> Int
```

Equations have a left- and a right-hand-side, e.g.

$$\underbrace{plusOne}_{\text{left-hand-side}} = \underbrace{(+)\,1}_{\text{right-hand-side}}$$

By using *pattern matching*, we can return different expressions depending on the parameters of the function:

```
sumTree :: Tree Int -> Int    -- Take a tree with Int as value
                              -- and return an Int.
sumTree (Leaf x) = x
sumTree (Node left x right) = (sumTree left) + x + (sumTree right)
```

Guard-expressions can be used to add further limit which equations will be used:

```
abs x | x<0 = -x
      | x>=0 = x
```

The right-hand-side of an equation can contain several different syntactical constructs. The most important are:

▷ `if-then-else`:
```
  if x==0 then "x is zero" else "x is non-zero"
```

▷ `pattern-matching`:
```
  case x of
    (Leaf _) -> "x is a leaf-node"
    (Node _ _ _) -> "x has child-nodes"
```

▷ local variable bindings using `let`:
```
  let x = 3 in x*x -- will evaluate to 9
```

▷ and the `do`-notation that is used for expressing side-effects (in the IO-Monad):
```
  do
    name <- getLine
    putStrLn ("your name is: " ++ name)
```

One important property of the grammar is that the indentation of source-code determines its structure. For example, for the *do*-notation, any statement that would start at the same indentation as `name <- getLine` would belong to the `do`-block

2.1.2 Logical Aspects of Curry

The first important difference to Haskell is that operations can be non-deterministic, i.e. they can return multiple results (or none). For example, the `?` operator non-deterministically returns one of its parameters; it could be defined as:

```
(?) :: a -> a -> a
x ? _ = x          -- first  equation
_ ? y = y          -- second equation
```

In Haskell, the first equation would always be used, because it does not contain any restrictions on its parameters. Therefore the second would never be used and thus would be superfluous. But in Curry, *both* equations can return a result.

Using this operator, we can for example define

```
coin = 0 ? 1
```

which would evaluate non-deterministically to both 0 and 1.

In non-logical functional languages, functions have a clearly defined direction of information: You cannot use the return value of a function unless you have provided actual values for its parameters. This means for example that you can not express the question "given a value v, what parameter do I have to provide to the given function f so that `f i = v`?" in Haskell code. *Logical variables* remove this restriction: It is possible to name, and consequently use, unknown values. In Curry we could write:

```
answer | f i =:= v = i where i free
```

As this definition might be strange to parse for anyone not familiar in Curry, Figure 2.1 describes the structure and semantics.

`answer`	`|`	`f i =:= v`	`=`	`i`	`where i free`
answer,	provided that:	$f\ (i)$ evaluates to the same ground data term as v	is	i.	(we need to label free variables a such.)

Figure 2.1. A semantic translation of the code for `answer`

`=:=` is an operator of type `a -> a -> Success` that defines an *equality constraint*. The difference to the equality operator == is, that the parameters may contain unbound, free variables. The type `Success` represents no real values, but expresses the result of constraints.

As a more concrete example, we can retrieve possible values of free variables by just stating the constraint at the Curry command-line. When we enter

```
[1, 2] ++ xs =:= [1, 2, 3, 4] where xs free
```

the system will bind `xs` to `[3, 4]`.

2.1.3 Ecosystem for Curry

There exist multiple different compilers for Curry, the main ones are:

- ▻ PAKCS [web:pakcs]: the Portland Aachen Kiel Curry System, targeting the logic language *Prolog*,
- ▻ KiCS2 [web:kics2]: Kiel Curry System 2, targeting Haskell,
- ▻ mcc [web:mcc]: the Münster Curry Compiler, targeting C.

See [web:curryimpls] for a more complete listing of implementations. For this project, we restrict the support to the first two compilers, KiCS2 and PAKCS. These two provide the same additional tooling that is used by the Curry IDE, namely the *Curry Analysis Server System* (CASS) [HS14] and the *Curry Object Observation System* (COOSy) [Bra+04]. The former provides static analyses about Curry programs and is, in the IDE, both made available to the user and used directly to retrieve type information. The latter is a debugging tool which can be accessed from the IDE as well.

Just like in Haskell, Curry has a standard module called "Prelude" which is imported by default even without a corresponding *import* statement. In addition to that module, both compilers provide a number of libraries. In this context, we call a module a *library*, if it provides common functionality or utilities for some topic and if it is usable in different applications. Example libraries are the *List* module, which contains some useful operations on lists, or the *Socket* module, which contains methods for network programming. A few libraries contain compiler-specific implementations and are only available for one compiler, but most are, with the same interface, available in both compilers.

2.1.4 Debugging in Curry

The Eclipse IDE was initially created for Java, and the core concepts for debugging in the IDE are aimed at *imperative* languages as a consequence. But, as Bernard James Pope put it:

> The traditional debugging technique of examining the program execution step-by-step, popular with imperative languages, is less suitable for Haskell because its unorthodox evaluation strategy is difficult to relate to the structure of the original program source code. [Pop06]

Which is also true for Curry, having a similar evaluation strategy with lazyness. For this reason, declarative languages use a different approach, called *algorithmic debugging* or *declarative debugging*. The core concept of this type of debugging is to log the flow of execution at run-time, allowing the programmer to observe *which* sub-expressions were evaluated or *what value* they evaluated to. The generated data is called *execution tree*. The amount of data can be rather large, and to isolate a problem, it is common that the debugger asks the programmer, who is called the "oracle" a series of questions about specific computations and their result, for example allowing the programmer to state "this result is correct", "this result is wrong" or "I do not know". [CS09]

For Curry, the compiler *kics* (not *kics2*) has integrated such a debugger, called *Believe in Oracles*.

A slightly different approach is represented by the *Curry Object Observation System*, which uses the same approach as the tool *HOOD* for Haskell [web:hood]: the programmer can select specific expressions in the source code (by inserting an `observe` function call). The evaluation of the expression and its sub-expressions is logged, so that the exact steps (what parts of the expression were evaluated, and to what values), can be seen afterwards.

It is also possible to use more basic ways of debugging in Curry: For example, the function `trace` from the library `Unsafe` can be used to print something as a side-effect whenever a certain expression is evaluated. In imperative programming, this approach might be called "printf-debugging".

2.2 Eclipse IDE

Eclipse is an integrated development environment (IDE) written in Java. It was originally developed by IBM, became open source in 2001, and is being controlled by the independent Eclipse Foundation since 2004 [web:eclwiki]. While initially being developed as an IDE for Java, it is now designed in a modular fashion that allows the underlying system to not only be used for different languages, but also for non-IDE software. Main example for the latter is *IBM Notes 8 and 9*, which provides business collaboration functionality (email, calendars, contact management).

The core concept that allow Eclipse's flexibility are *components*. For Eclipse, these building blocks are often called *plugins*; applications consist of a number of components. Components can depend on, and consequently access, other components. To allow continuous upgrade of software, components have specific versions.

For the Eclipse IDE, the contained components can be roughly partitioned into a few groups. At the base stands the *Eclipse Rich Client Platform* (Eclipse RCP), which is based on the Java Runtime Environment and provides application-independent functionality, e.g. the graphical user-interface toolkit *swt*. Also, Eclipse RCP contains an implementation of the OSGi framework specification [web:osgi], which manages the composition and interaction of different plugins.

On top of Eclipse RCP, the *IDE Platform* contains the plugins that form the language-independent base for the IDE the term *Eclipse* commonly refers to. Furthermore, the *Eclipse SDK* contains a wide variety of tools used by different plugins. [ML05, p. 3-25]

2.2.1 Source Files, Projects and Workspaces

On a basic level, a *project* is a directory on the file-system that contains source files (for some arbitrary language) and/or configurations (such as xml files). As such, projects are a means to provide structure to elements that implement a certain functionality. Example projects would be a Java application, that consists of a tree of classes, one of which is the main class of the executable; or a feature project, which is a project containing basically only one configuration file naming the *plugin projects* that constitute the *feature*.

In many cases, multiple projects are semantically connected or even function as a unit. To allow the user to work on multiple projects at once, Eclipse uses the

```
Java Development Tools   Curry IDE
|            |            |  |
|         /--+------------/  |
Eclipse SDK  |               |
|     |      | /--------------/
|   IDE Platform
|     |
Eclipse RCP
      |
(Java Runtime)
```

Figure 2.2. Component Groups and Dependencies in Eclipse

concept of a *workspace*. A workspace contains a list of projects. On the file system, the workspace is a directory that contains the description of the contained projects, plus any workspace-specific configurations that the user can make. Projects can be located inside the workspace directory, but they can also be *external projects*; in the latter case, the workspace contains just a reference to the actual project directory.

When the Eclipse IDE is started, the user has to choose the workspace directory that will be opened. That means that exactly one workspace is opened at a time. File-system locks prevent the same workspace to be opened by different Eclipse instances at the same time.

If a project has specific contents, we call it a "*foo*-project". For example we might call a project containing a java executable "Java project".

It should be noted that the *git* repository for this thesis contains the project directories, *not* the workspace directory. To start developing, these projects must be imported into a separately created workspace. The details are described in Appendix D.

2.2.2 More Terminology: Projects, Plugins, and Features

The term *plugin* is has multiple meanings in the context of the Eclipse IDE. We would commonly call the Curry IDE a *plugin* for Eclipse. Yet, the correct technically term for the complete collection of extensions that form the Curry IDE is *feature*. A feature can bundle multiple plugins, or rather: multiple *plugin projects*. For example, the Curry IDE *feature* (currently) consists of three plugin projects.

Each plugin project has a file `plugin.xml` in its root directory that describes the plugin's properties such as the version, its dependencies and both usage and provision of extension points. The feature is declared in a separate project as well, which contains a `feature.xml`.

Assuming that we have a finished feature project and a number of plugin projects, there still is one more step necessary in order to allow an Eclipse user to install the feature in the IDE: We need an *update site*, which contains the compiled packages for one or more features. When installing new features, the user can choose an update site. Eclipse then downloads the information about available features, and then the use can choose specific features to install. In many cases, these update sites will be online, for example the "plugin" for Scala can be installed from `http://download.scala-ide.org/sdk/e38/scala210/stable/site`.

In our case, there is an update site, but we will use the update site as a local resource.

2.2.3 Plugin Details and Extension Points

Extension points are the interface that connect different plugins. By defining a new extension point, a plugin allows future customization. By using existing extension points (i.e. by providing implementations for some specific extension point), a plugin can add new implementations or configurations.

Lets consider a basic, yet abstract, example:

We are programming a plugin A and want to allow future customization of feature X. To realize that, we statically define an extension point X, with a suitable interface. At run-time, we request a list of implementations for X from the extension registry, and use one or more of these.

For this example, the extension point will contain two items: One configuration value of type `String` and one of class-type (represented by the class's full name) extending a specified interface.

"Using" in this context could simply mean that we read some values from the extension definition. the extension point could, however, also contain some fields that are Java classes (or some other means of obtaining instances of some interface at run-time); then, we could instantiate and call methods on these classes. This way, extension points are the entry points, control-flow-wise, into new plugins.

```
plugin-project A
+ XInterface.java                               // Java interface used
|   *------------------------------*            //    in the extension
|   | public interface XInterface { |
|   |   public void foo();         |
|   | }                            |
|   *------------------------------*
+ plugin.xml                                    // the extension "declaration"
|   *-----------------------------------------*
|   | ...                                     |
|   | <extension-point id="X" schema="X.exsd"> |
|   | ...                                     |
|   *-----------------------------------------*
+ X.exsd - XML schema file                      // the extension "definition"
    defines a schema with the following members:
      myConfigString: String
      myConfigImpl: String, must be the
        qualified name of a class
        implementing XInterface
```

Figure 2.3. The static setup of project A, that provides extension point X

```
plugin-project B
+ XImplementation.java                          // the implementation
|   *-----------------------------*
|   | public class XImplementation |
|   |   implements XInterface {    |            // note that A.XInterface is
|   |   public void foo() {        |            //   known
|   |     ...                      |
|   |   }                          |
|   | }                            |
|   *-----------------------------*
+ plugin.xml                                    // registration of
    *-----------------------------------------*  // the implementation
    | ...                                      |
    | <extension point="X">                    |  // not "extension-point" (!)
    |   <myConfigString value="hello, world">  |
    |   <myConfigClass class="XImplementation" |
    | ...                                      |
    *-----------------------------------------*
```

Figure 2.4. The static setup of project `B`, that implements extension point `X`

To extend the functionality of *A* for *X*, we would add an extension for *X* into a new project *B*. Assuming that plugin *B* is installed, the extension registry will list the respective entry under *X* at run-time. Furthermore, when *A* calls method in this implementation of *X*, *B*'s code will be called.

For this example, the static setup, i.e. the files involved and what they contain, is shown in Figures 2.3 and 2.4. For the schema file we summed up the contents, because actual the notation (which uses xml) is too long. Note that project `B` has a dependency to project `A`, as the `XImplementation` implements the `XInterface`.

To give an example of the events at run-time (Figure 2.5), we assume that by some undefined means the control flow is in plugin A (e.g. because it implements one of the extensions provided by Eclipse). In plugin A, we use Eclipse's *extension registry* to retrieve the list of implementers for extension point `X`, and use the first one (with `[0]`). First we read the `String` property, then, we request an instance of the *class* property, and call a method on that instance.

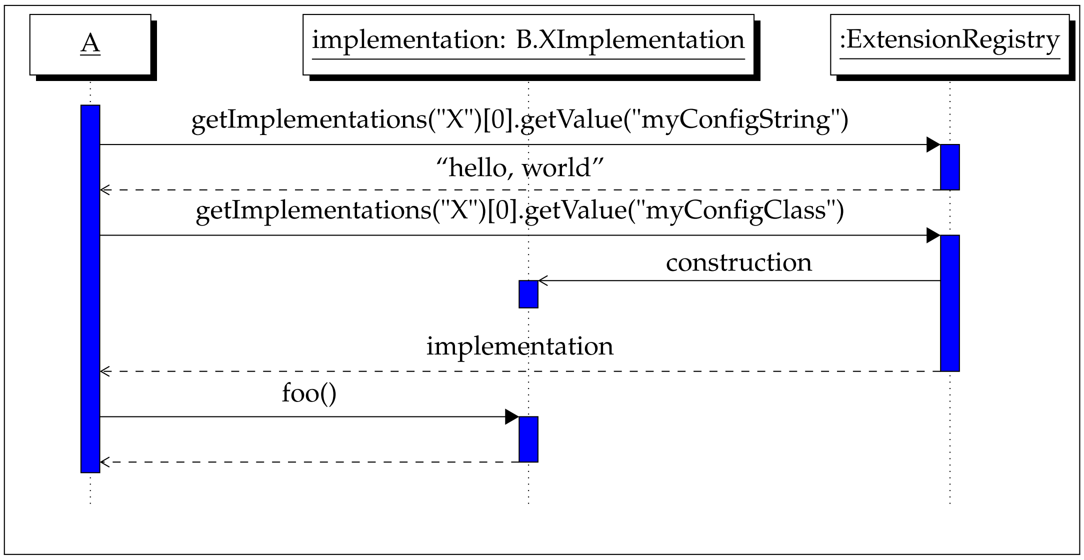

A
implementation: B.XImplementation
:ExtensionRegistry
getImplementations("X")[0].getValue("myConfigString")
"hello, world"
getImplementations("X")[0].getValue("myConfigClass")
construction
implementation
foo()

Figure 2.5. The run-time events involving the extension point X

In most cases, we will use existing extension points (e.g. the ones provided by Eclipse or by Xtext). In those cases, our code will contain only the items described in project B, as the extension point is provided externally.

2.2.4 User-Interface Concepts

Typically, the interface of the IDE will show multiple sub-windows, each with some specific functionality. These are called *views* and can be re-sized and re-arranged freely. Typical views are:

▻ The project explorer provides a structured view of the workspace contents, i.e. a list of projects and the folders and files inside.

▻ The editor contains a tabbed list of editors for specific files. Depending on the type of file, this can be a simple text editor, a more advanced text editor with language-specific features such as highlighting, a custom form with editable elements or a some form of custom graphical editor.

  For the Curry IDE, the Java/Scala source code will be shown in an advanced text editor; the project's manifest[2] is presented as a series of forms.

[2]right click a project in the project explorer, plug-in tools, open manifest

▷ The *Problems* view shows warnings and errors regarding the files in the project. For example when a Java file contains syntax errors, the error will be both highlighted in the editor window and listed in the *Problems* view.

▷ The *Console* view integrates a terminal into the IDE. Using this view, the programmer can access various command-line applications, such as the Scala REPL.

▷ The *Outline* provides a structured summarization of the contents of the editor window which is currently active. For example, for a Java class, it might contain nodes for the imports, the data members, and the public and private member functions in the class.

Some of these tools are language specific or task-specific. A view for displaying Javadoc is only usable for Java; Debugging utilities such as a view showing variables and their current value are connected to the task of debugging. For this reason, the Eclipse IDE has *perspectives* that define a list of relevant views as well as the arrangement of these views. One view is active at a time, and the user can switch freely between them. Apart from views, certain other elements of the user-interface, for example the elements of context menus, can be configured to be specific to a certain perspective.

Typical perspectives are the *Java* perspective (used when editing Java code) or the *Debug* perspective (which contains the common tools for debugging imperative programs). For the Curry IDE, Marian Palkus introduced the *Curry* perspective which contains any Curry-specific views.

2.2.5 Building and Launching

The topic of launches is relevant in two instances: Firstly, when developing the Curry IDE, we launch the plugin; secondly, the user of the Curry IDE launches Curry programs from the Curry IDE. There are different kinds of launches: Launching a Curry program (which opens a Curry console) is different from launching a Java application for debugging. For this reason, launches have two parameters:

▷ A specific launch *configuration*: Launch configurations have a type, e.g. *MWE2 workflow* or *Curry Application*, that will determine what is launched. Additionally, the launch configuration can contain settings that define details about the launch, e.g. the run-time options or the `Main`-class to use for a Java application.

- ▷ The launch *mode*: Typical modes are *run* and *debug*. Launch configurations may be able to handle different modes; for example in Curry the same launch configuration will work for the *run* and *debug* modes; in both cases opening a console but in slightly different ways (behind the scenes) to make debugging possible.

From the user's perspective, in the workflow of a launch, the *mode* is chosen first: For example, the context menu on a project might contain the sub-menus *launch as* and *run as*. When using the *configurations* dialog in these menus, the list will be sorted by configuration type, and then contain the concrete configurations. Of course, there are shortcuts to this process; for example the user-interface contains *run*- and *debug*-buttons in the toolbar which will re-execute the last launch, i.e. the *run* button will execute the last launch configuration in *run* mode.

By using Eclipse's extensions, it is possible to define both new *modes* and new *configuration types*. As we already mentioned, the latter is used to implement launching of Curry programs.

2.3 Xtext

Xtext [web:xtext] is an open-source framework for Eclipse for the integration of new languages into the IDE. It uses the parser-generator *ANTLR* [web:antlr] to create a parser from the grammar of the new language, and includes by default features like syntax checking and syntax coloring. This functionality can be customized for adding custom features such as:

- ▷ content assist: Provides code completions to the user of the editor;
- ▷ validations: In addition to the syntax checking done by the parser, custom semantic checks on the resulting syntax tree can be implemented;
- ▷ code formatting: automatic formatting of the source files

The syntax tree generated by the parser is represented as Java objects, the corresponding classes are automatically generated from the underlying grammar using the *Eclipse Modeling Framework* (EMF).

2.3.1 Static Structure of an Xtext Project

By default, an Xtext project consists of four projects; we will assume that the main name is “foo”:

foo	core language implementation, such as the grammar definition and semantic checks
foo.ui	user interface, such as the *contentassist* implementation and the project *outline*
foo.tests	test code
foo.sdk	the *feature* project that bundles the other three; no sourcecode

The first two projects have a number of predefined packages; the corresponding class files are separated into multiple directories: Firstly, there is the usual `src` directory, secondly there is `src-gen`. The `foo` project contains a third folder `xtend-gen`. The main idea of this separation is that the `src` folder contains code written by the plugin’s developer, while the other folders contain auto-generated code that should not be modified by the programmer (as it will be overwritten when building the project). Under certain circumstances, for example when deleting certain modules or just after creating a new project, empty classes will be generated in the `src` folder, but they will not be automatically modified if they exist (which might be confusing at first).

2.3.2 Generation Workflow

For many simpler projects, creating the usable, compiled package is as simple as compiling and bundling all the java files in the source directory. For Xtext, the process is more complex, because it involves automatically generating source code at multiple places:

- the parser is generated from the grammar using ANTLR (ANTLR produces java code),
- the language model, i.e. the classes representing our syntax tree, are generated from grammar using EMF,
- Xtext supports a dialect of Java called Xtend [web:xtend], that supports certain features like macros, lambdas and operator overloading which are not present in Java. This dialect is compiled to Java. In the Curry IDE, we do not (actively) use this dialect, though.

To manage the code generation, Xtext uses the *Modeling Workflow Engine 2* (MWE2). The core project will contain a `.mwe2` file that describes which code is generated. Depending on the features that the IDE should support, certain elements, called *fragments* can be inserted or omitted. Example fragments are the parser generator or the quickfix provider.

This workflow must be explicitly executed by the plugin's developer, i.e. it is not part of the Eclipse project build process. But unless the workflow or the grammar file changes, it is not necessary to re-run the workflow. For the details of how to completely rebuild the plugin, see Appendix D.

2.3.3 Google Guice

Xtext uses the dependency injection framework *Google Guice* [web:gguice] to manage the dependencies between the different classes.

One simple example to understand what dependency injection does:

To allow customization, we define an interface `MyInterface`. In class `Foo` we need an implementation of said interface. Without dependency injection we have two options:

- ▻ `Foo`'s constructor takes a parameter of type `MyInterface`.
- ▻ `Foo` instantiates a given implementation of `MyInterface`.

Both options have the same disadvantage: We need to statically *resolve the dependency*, i.e. connect an implementation of `MyInterface`, either directly in `Foo` or indirectly when constructing `Foo`.

Dependency injection separates the dependency resolution from the implementation (in this case, from *Foo*). We can write:

```
class Foo {
  @Inject
  private MyInterface providedImplementation;
}
```

where the semantics of the `@Inject`-annotation basically is "Guice, provide me, at run-time, with an implementation of `MyInterface`". Then, in a separate location (and independently from `Foo`), we define the dependency resolution, i.e. we tell Guice what implementation to use for which interface.

In the case of Xtext, the dependencies are resolved for the core- and the ui-projects in the `RuntimeModule` and `UiModule` classes. (For the Curry IDE, in the file `CurryRuntimeModule.java` and `CurryUiModule.java`). These files are inspected using reflection to find the different `bind...` methods; each `bind` method defines the implementation to use for a certain interface.

It is also possible to directly inject classes (not interfaces); in this case, it is not necessary to define a binding.

We can use:

```
class Foo {..}
class Bar {
  @Inject
  private Foo foo;
}
```

On its own, this might seem useless; we could just as well omit the `@Inject`-annotation. But we can use the `@Singleton`-annotation to change the semantics: When `Foo` is annotated with it, only one instance will be created[3], even if we inject `Foo` in multiple places or create multiple instances of `Bar`.

When we write:

```
@Singleton
class Foo {..}
class Bar {
  @Inject
  private Foo foo;
}
```

[3]technically, one instance per *injector*; see [web:ggsngltn]

```
class Baz {
  @Inject
  private Foo foo;
}
```

Then the same instance of `Foo` will be injected into instances of `Bar` and `Baz` at run-time.

When injecting, Guice will work recursively: If the injected class contains `@Inject`-annotations, those will be injected as well.

if `Foo` from above contains any `@Inject`'ed members, they will be injected as well when constructing `Bar` or `Baz`.

In the Curry IDE, any classes that make use of injection are created directly or indirectly from extension points. In these cases, the extension point definition will contain an indirection (using a so-called `ExecutableExtensionFactory`) that triggers the injection. If the user would want to use injection for manually created instances, he could request an *injector* from the plugin's configuration classes and apply it.

2.3.4 The Xtext Grammar File

The core project contains a `.xtext` file (here: `Curry.xtext`) that defines the grammar of the language. This file has the purpose of

1. defining the language being recognized by the parser;
2. specifying how the syntax tree is created, i.e. what is the name and what are the members of the different classes representing the syntax tree nodes;
3. separating the lexer and the parser (both of which are created from this file);
4. defining reference relationships between identifiers in the language. In many languages, identifiers can be defined at one place, and then used in a certain scope. Xtext supports a way of resolving these references as part of the parsing, creating effectively a syntax graph instead of a syntax tree. The blog at [web:xtextrefs] explains this functionality and its syntax in more detail.

The grammar file contains a list of rules that are similar to EBNF rules, but with certain additional annotations to allow these additions.

The rule for signatures in Curry IDE’s grammar is

```
Signature:
  {Signature} functions=FunctionNames '::' type=TypeExpr;
```

To understand this rule, it might be useful to compare this to the corresponding pure EBNF:

```
Signature ::= FunctionNames '::' TypeExpr;
```

The additions in the Xtext grammar define how the Java interface (and its members) generated for each rule are called. In this case, the generated interface looks like this:

```
public interface Signature extends FunctionDeclaration {
    FunctionNames getFunctions(); // FunctionNames and TypeExpr are other
    TypeExpr      getType();      // interfaces representing the subnodes.
}
```

I do not want to go into further details of the grammar file, as the grammar was not the focus of this thesis (even though it was modified in a few places). One property of the grammar is worth mentioning, though: Rules that contained non-zero lookahead, i.e. rules with multiple alternatives that could start with the same terminal, seemed to cause problems. Certain rules, like the different list literals (`[]`, `[a]`, `[a,b]` `[a..b]`, `[a,b..c]`), are a bit more complex for that reason.

2.4 CASS

The *Curry Analysis Server System* (CASS) is a tool for the static analysis of Curry programs. CASS is generic: Various kinds of analyses (groundness, non-determinism, demanded arguments) can be integrated. In order to analyze larger applications consisting of dozens or hundreds of modules, CASS supports a modular and incremental analysis of programs. [web:cass; HS14]

There are multiple ways of accessing the tool’s functionality:

- ▻ In batch mode, using command-line parameters to trigger specific analyses, and showing the results to the user in the console;
- ▻ In API mode, where the tool is used as a library in a Curry program;
- ▻ In server mode, using TCP sockets and a simple communication protocol.

When using CASS, the Curry IDE uses the third possibility.

2.5 COOSy

The *Curry Object Observation System* is a tool for observational debugging Curry programs. Currently, the tool only works with the *pakcs* compiler. It consists of a library that will be imported by the program being to be debugged, and a viewing tool, i.e. simple user interface displaying the observations. [web:coosy; Bra+04]

Observing an expression in the sense of COOSy means following the evaluation of that expression. Because of the lazyness in Curry, only parts of the expression might ever get evaluated. The important property of COOSy is that it does not force the evaluation of the expression, while still showing the values of any sub-expressions that are evaluated.

For example, when observing, using COOSy, the infinite list `[1..]` in the expression

```
print $ take 3 $ [1..]
```

the observation view will show `1:2:3:_`, the underscore denoting that the rest-list was not evaluated. If we had used some form of *printing* to observe the expression, the observation would have to print the infinite list and would not terminate.

When running the program containing observations, the evaluation of the relevant expressions will produce a series of messages that are written to files in a newly created directory `COOSYLOGS`. The viewing tool will read and interpret these files to show the observations to the user. See Appendix C.5.2 on how to use observations.

Because Curry has no type classes, it is currently necessary to add a form of typing information when adding observations.

For example the code of the observation added in the example above would be:

```
print $ take $ observe (oList oInt) "infinte list" [1..]
```

`oList` and `oInt` are functions describing how the debugging for the respective types works, and `[1..]` is a list of `Ints`. The string literal serves as an identifier for this observation.

Note that, judging by the program's semantics, the `observe` function acts as the identity on its third argument. The need for information on the type the observed expression will be important when adding observations as part of an automated process (which we implemented in the IDE).

2.6 Scala

Scala [web:scala] is a statically typed, object-oriented and functional programming language with a close relationship to Java. It compiles to Java bytecode and therefore runs on the JVM, and Scala and Java code can interoperate, i.e. Java libraries and classes can be used in Scala, and vice-versa. Scala is an acronym for "scalable language", meaning that it can grow with the programmer and his/her needs. [OSV08]

For a Java programmer, Scala may be initially seen and used as "a Java without semicolons", as the general project structure and syntax of the files is similar: Packages, their definition, imports, and the general form of class definitions with curly braces are all almost unchanged. But Scala both adds constructs from functional programming and tries to improve in object-oriented aspects.

The main features from functional programming are:

▻ First class functions and anonymous functions;

▻ Type inference - but naturally there are limits for the inference for a type system with an object-oriented nature with classes and inheritance;

▻ Algebraic data types and pattern matching;

▻ Support of tail recursion;

▻ Immutability: for example, for many data structures, both mutable and immutable variants exist.

To manage builds, Scala provides the *Scala build tool* (sbt), that supports incremental rebuilding. This tool is used by Eclipse when building Scala projects.

Chapter 3

Old and New Features

In this chapter, we would like to give an overview of the features we implemented. Being based on somebody else's work, it is important to keep track of which features were already implemented before we started our work and which features are actually new. Consequently, the next subsection will highlight the previously existing features; after that the new features will be described.

3.1 Existing Features

3.1.1 Basic IDE features

Being implemented as a plugin for Eclipse, the Curry IDE does provide all the standard features of Eclipse that are not specific to a programming language. This includes:

▻ a graphical user-interface with customisable tool-bars and multiple sub-windows that can be freely rearranged. Sub-windows for specific purposes are called *views*.

▻ a project explorer is a view providing an overview of the file-system directory structure of the project; in the case of Curry this project folder will probably contain a `src` (source) sub-folder containing all the Curry modules that constitute an application or library.

▻ one or multiple tabbed editor windows can be opened to view and modify the project's resources. In the case of Curry, the resources are Curry modules, so the editors are text editors.

▻ The "problems" view with the purpose of listing errors and warnings in the project's resources. For example, syntax errors would be listed in this view.

When editing, Eclipse does also provide expectable basic editing functionalities like

▻ a editing history, i.e. the ability to undo and redo changes (shortcut: ctrl-z).

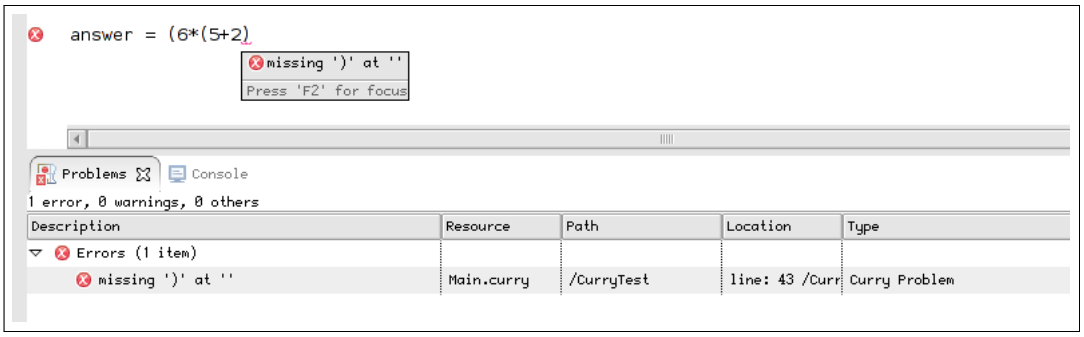
▻ searching and replacing, both for single documents and project-wide (or even workspace-wide, where the *workspace* contains a set of projects that the user currently works on).

▻ bookmarks that allow the user to add personal notes in his files

Now let us have a look at the features specific for Curry, which were added by Marian Palkus:

3.1.2 Syntax Checking

The Curry IDE executes a syntax check for all opened Curry source files (Figure 3.1). This syntax check does not use any external tool (i.e. any existing Curry interpreter/compiler). The advantage of having a separate implementation of the parser is that this check is lightweight and can be executed in the background while the user edits, giving him/her instant[1] feedback about such errors.


```
answer = (6*(5+2)
```
```
missing ')' at ''
Press 'F2' for focus
```

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
missing ')' at ''	Main.curry	/CurryTest	line: 43 /Curr	Curry Problem

Figure 3.1. Screenshot of the syntax error representation. Visible are two views: At the top is the editor view containing the code with the syntax error (by hovering over the underlined text the pop-up was opened, displaying the error message). At the bottom, the *Problems* view lists any errors or warnings in the opened projects.

Because the IDE has access to the syntax information, certain features are available in the source editor:

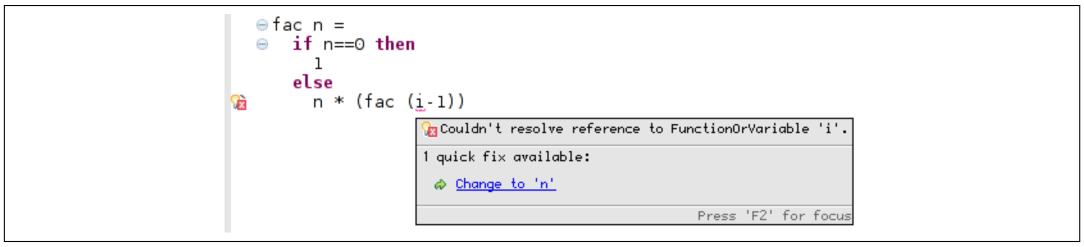
[1]where *instant* means “in less than a second”

▷ Basic syntactic highlighting: For keywords and operators, the user can choose highlighting colors. However, this highlighting is rather basic, as it only uses lexical information at the moment; also it does not distinguish comments and literal in the code.

▷ Code folding: Sections of code belonging to the same syntactic construct (for example a function and its whole body) can be folded into one line. This might be helpful when examining large source files (or files with large functions).

3.1.3 Linking and Scoping

The Curry IDE does not execute a full static program validation (i.e. it is possible to write Curry modules which pass all checks of the Curry IDE but which still will not compile using an actual interpreter/compiler). However, it also does more than just syntax checking: it does check the cross-referencing (the *linking*) of identifiers acknowledging the visibility of the referenced symbols in the code. Figure 3.2 shows such an error, where we mistakenly used the identifier `i` instead of `n`.



```
fac n =
  if n==0 then
    1
  else
    n * (fac (i-1))
```

```
Couldn't resolve reference to FunctionOrVariable 'i'.
1 quick fix available:
  Change to 'n'
                                   Press 'F2' for focus
```

Figure 3.2. Example of a cross-referencing error

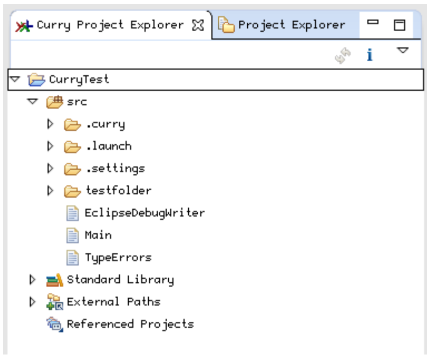
3.1.4 GUI elements

One Curry-specific view, the *Curry Project Explorer* (Figure 3.3) was added to Eclipse. It is intended to replace the default, language agnostic project explorer provided by Eclipse. The *Curry Project Explorer* contains two configuration items:

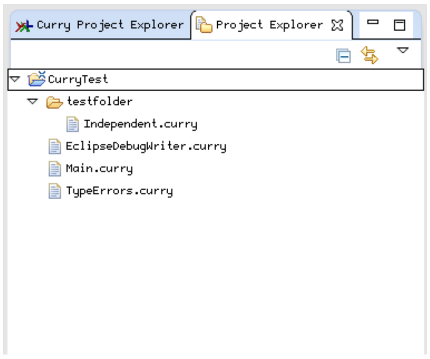
1. the Curry standard library path. It should be defined so that the module files from the standard library can be found and so the corresponding symbols can be resolved.

2. the *external paths* option, which defines a list of paths with external[2] Curry

[2]external meaning "belonging neither to the standard library nor to this project"



(a) The Curry Explorer, added by Marian Palkus



(b) as a comparison, the default Project Explorer provided by the Eclipse IDE

Figure 3.3. the Curry Explorer

modules that can be used in this project.

Also, the interface includes two “wizards”, i.e. dialogs with the purpose of creating new resources (projects or modules).

In order to execute the Curry program from Eclipse, an “external tools” configuration template was added. This can (in theory[3]) be used to start the external Curry interpreter in a console view. This console view would then be usable like a terminal window running an instance of the interpreter.

Unfortunately, the *Curry Project Explorer* is unfinished and provides, as it is, no real benefit over using the default project explorer. The main problem is that certain error conditions are not handled properly, so the user is presented some indecipherable message.

3.1.5 Curry Analysis

One external tool that was planned to be accessible from the IDE is the *Curry Analysis Server System* (CASS) which provides static program analyses (see Section 2.4 for a more detailed description). This tool was still being developed while our predecessor worked on his thesis. As a consequence, the integration of this tool

[3]I personally did not manage to find the correct preconditions to actually launch the interpreter in a console window

is not complete. While from the users point of view, the *Curry Analysis Server System* was not usable at all, behind the scenes a lot of the work had indeed already been done. A menu point containing the different options of analyses was already prepared; however, it was not visible because the connection to the analysis server was not configured yet. The logic to request a specific analysis and to receive the result was implemented as well.

Apart from the connectivity problem, the only feature missing was some form of output of the result to the user.

3.2 New Features

3.2.1 Curry Analysis

Both aforementioned problems preventing the user from actually using CASS were resolved. Firstly, the connection to the analysis server was implemented. If the analysis server is running, the user can right-click elements in the Curry module's editor. The context menu which is displayed will now contain one item for the analyses. Using the sub-menu, the user is able to select a specific analysis, which is then executed externally. Secondly, a functional, albeit simple, output for the results of all analyses was added to the user interface. This output is called the *Curry Analysis View*.

A sample workflow for using CASS from Eclipse is:

1. The user starts the CASS executable (the "analysis server");
2. The user starts the Curry IDE (Eclipse plus the Curry plugins);
3. The user opens a Curry module;
4. The user opens the context menu on some element of the module, and selects a specific analysis;
5. The analysis is executed externally;
6. The results of the analysis are retrieved by Eclipse and displayed to the user in the *Curry Analysis View*.

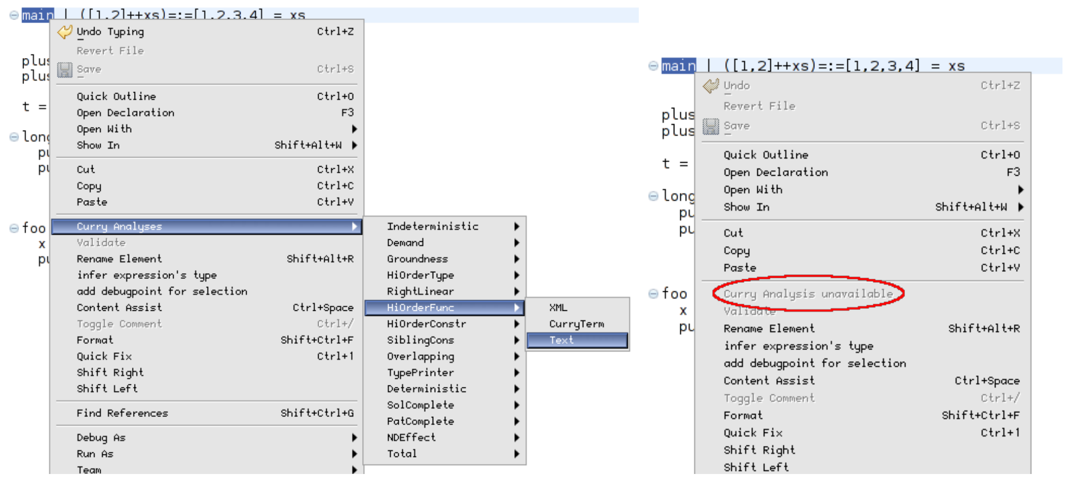
From time to time the user might forget to start the analysis server before starting Eclipse. In such a case the analyses of course are not available. In such cases, the menu entry for analyses now is not removed, but grayed out instead, which is more consistent and after all should be more clear to the user. Figure 3.5 shows the context menu for the two cases, i.e. if the analysis server is available and if not.

3.2.2 Curry Console and Error Annotations

As described by our predecessor, the development process involves both an editor and an interpreter/compiler, and a good amount of switching between the two. For two reasons this remains true even with the Curry IDE:

1. There are certain kinds of compile-time checks that are not implemented in the IDE. So (even) if there are no problems detected by the IDE, the user will inevitably need to make use of an actual compiler/interpreter.
2. While successful compilation is often a positive hint, this does not guarantee the program does what was intended. To check that, the user needs to execute the program (or parts thereof), and he consequently needs the compiler/interpreter.

The existing "external tools configuration" in combination with the *console view* were supposed to address this requirement. The respective code was to a large part rewritten to make this feature more usable and stable. The resulting changes and additions can be summarized as follows:



(a) with running CASS, showing the choice of analyses **(b)** when CASS is not running

Figure 3.5. the Curry Analysis Context Menu

1. In the Palkus IDE, the launch configuration for Curry used the "external tools" launch mode. This was changed; the *run* and *debug* modes are used instead of *external tools*. This change is intended to improve usability: On the one hand, "run/debug" is easier to access in the default Eclipse interface. On the other hand, run configurations are generally [4] used to execute the code being edited in Eclipse.

2. The configuration items (the run-time command to use for the launch, among other things) remain mostly unchanged.

3. The console view remains unchanged from the user's perspective. However, there is an additional feature:

4. All output from the Curry interpreter is analyzed. Compilation errors are detected and parsed; specifically the position (file and line number) of each error is determined. This information then is used to add a "problem marker" to the respective document.

3.2.3 Typechecking/Typeinference

Fast feedback about compile-time errors in the source-code is an essential part of any IDE. The obvious way to improve the IDE in that respect is to enlarge the number of classes of errors being checked. Typeinference is an interesting candidate: It had previously not been implemented at all in the Curry IDE and, assuming that the programmer is familiar with Curry and avoids most syntax errors, type errors will probably form the largest group of (compile-time) errors encountered.

Because the amount of work for (re)implementing[5] the full typeinference algorithm is rather large, a compromise was made between the two goals: On the one hand to detect as many of the common type errors in the IDE, while on the other hand keeping the size and complexity of the implementation manageable. So for certain cases the IDE will "miss" type errors. While there are *false negatives*, *false positives* are avoided, i.e. the IDE will only in very few cases report errors if the source code in its current state does in fact compile without a problem. The details about which errors are detected are difficult to explain without a closer look about the

[4]The plugins for Java, Scala and C++ use *run/debug* to launch projects, not *external tools*.

[5]*re*implementing, because the full typeinference algorithm is already implemented in the existing compilers

implementation, which is described in Section 4.4; the specific conditions that can lead to false positives are described in Section 4.4.1.

When a type error is detected, an *error marker* is added to the respective resource. The marker contains a description of the error, just like a corresponding message from the compiler would. The IDE uses a custom implementation to check for type errors. We tried to implement the algorithm in such a way that the error messages are understandable and provide as much information as possible to the user. Still, the messages might be slightly less understandable than the output of the compiler (especially for certain complex cases).

Apart from checking for errors, the typeinference implementation was used to add one new utility: The user can select any expression in the edited program and infer its type. For this purpose, a new view, titled "inferred types", was added to the user-interface.

3.2.4 Debugging

Even when a program compiles without any errors, it might not do what the programmer expected. To remove such deficiencies, the programmer has to debug the program. In rare cases the programmer might be able to fix the bug just by looking at the source-code again, but in general it is really helpful to analyze the details of what exactly is happening at run-time. The IDE can provide to the programmer *debugging tools* with the purpose of gathering data at run-time.

The Curry IDE was extended by a group of features that allow the programmer to trace and observe the run-time evaluation of arbitrary expressions in the Curry program[6]. From the user's point of view there are three additional features, which will be introduced in detail:

1. Curry *debugpoints* determine the expressions to be debugged. The IDE's interface contains a new view to manage debugpoints.
2. A specialized implementation of the Curry launch configuration for the *debug*-mode
3. the *Curry Trace Debugging* view displays the results of debugging (when using *trace* or *traceValue* (see below) as the debugpoint's type)

[6]See chapter 2.1.4 for a general introduction about debugging in Curry

Debugpoints are the equivalent of *breakpoints* known from imperative debugging. A *breakpoint* typically consists of a location (usually a file and line number) and certain modifiers that describe the action if the breakpoint is reached at run-time. Similarly a *debugpoint* consists of a reference to an expression in the source-code (i.e. the location) and the type of *expression debugger* that will be executed when the expression is evaluated at run-time. Considering the similarities between the two, it might seem straightforward to reuse the existing functionality for breakpoints being provided in Eclipse. Unfortunately, the two types are incompatible, because *breakpoints* refer to lines in the source-code, whereas *debugpoints* refer to expressions. The *Curry DebugPoints* view was added to the interface. It contains a list of debugpoints for the opened projects. Debugpoints are persistent, i.e. closing and reopening Eclipse will not clear the list. To add a new debugpoint, the user can select an expression in the source code and use the corresponding item in the context menu. In the list, the user can change the type of debugpoint. At the moment, there are four types:

1. null: does not do anything. This type exists as a minimal example for the IDE's programmer; it is not supposed to be useful to the IDE's user.
2. trace: notifies the user each time the resp. expression is evaluated, but does not capture the value in any way.
3. trace with value: notifies the user each time the resp. expression is evaluated and prints the value of the expression. Note that printing the value can cause additional evaluation of sub-expressions. See figure 3.7 for an example.
4. observation of evaluation: using the external debugging tool *COOSy*, all evaluations of the resp. expressions and its sub-expressions can be captured.

A specialized **debug launch** is added to the Curry IDE. Just like the normal launch, this launch opens a console window running an interpreter. The difference is that the interpreter will run a separate environment containing a modified version of all the Curry source files in the project. Each debugpoint causes some modification to the source code to implement the specific behavior. While behind the scenes, the *debug launch* does the additional preparations described above, the only difference to the user should be the feedback from the debugpoints[7].

[7]There are a few special cases where a difference is noticeable: If the program does not compile, the error message might be different when using debug launch. When "escaping" from the interpreter

```
debugPrint :: String -> a -> a
debugPrint ident x = unsafePerformIO $ do
  putStrLn (ident ++ " " ++ show x)
  return x

main = print $ take 3 $ debugPrint "longList" [1..1000]
```

Figure 3.7. `debugPrint` is a simple implementation of a basic debugging function that acts as the identity, but as a side-effect (using `unsafePerformIO`) prints the value of its second parameter. While without the debugging function, only the first three items in the list would be evaluated, the `show` function will evaluate the complete list. One can imagine that this is undesirable, especially considering that the list might be not just large, but infinite.

To display the results of the trace debuggers (both with and without value), the **Curry Trace Debugging** view was added to the interface. This view contains the chronological list of traces encountered during run-time.

There is no output integrated into the IDE for the *observation* debugpoint type. Instead, the user must open the external tool *COOSy*. The process for this is briefly described in Appendix C.5.2; the COOSy GUI is introduced in [Bra+04]. Also, it should be noted that the implementation of the debugpoint types provides a generic interface so that new types can be embedded easily in the future.

to the system shell, for example using ":!pwd", the user can observe the changed environment. This trick can be used to display the modified source-code, too.

Chapter 4

Implementation

4.1 Structure of the Project

Comparing the high-level structure of the source code my predecessor passed on to me and the current version, there are some significant changes. This affects the implementation of all of the features; therefore we would like to describe the structure now. There are six Eclipse projects. These are:

CurryIDE[1]	source code	core Xtext implementation
CurryIDE.ui		user interface Xtext functionality
CurryIDE.uinonxtext		non-Xtext user interface functionality
CurryIDE.tests		test-code
CurryIDE.sdk	formal project description	bundles the other projects into a *feature* project
the *update site* project		allows the *feature* to be installed in Eclipse

Because of the way the Eclipse builder works, dependencies between projects must form an acyclic graph. the order of the projects in the table reflects this to some degree: all dependencies go from lower to upper projects. The project *CurryIDE.uinonxtext* is new and contains source code that is not connected to Xtext directly. An example would be the implementation of the launch and the Curry console, as these two use Eclipse's extension points, not those of Xtext. The reasoning behind the changes to the structure is given in Section 4.6.1; a more detailed listing of the projects and their packages is in Appendix E.

[1]the complete name is *de.kiel.uni.informatik.ps.curry.CurryIDE*; the prefixes were omitted for brevity

4.2 The Analysis View

In order to make the infrastructure provided by Palkus for the Curry Analysis Server System (CASS) available to the user, we can identify three separate modifications or additions.

The first change was to fix the connection to the analysis server. The TCP port used when connecting to the server is read from the `.curryanalysis.port` file in the user's home directory. This file will be created by the server as soon as the server is started. A second problem was the lifetime of the connection: The analysis server previously closed the connection after each handled request, while the Curry IDE (i.e. the `CurryAnalysisToolClient`) expected the connection to stay open for further requests. This caused the analysis integration to fail after the first request, which always was the request for listing all available analysis types, done automatically at the start of Eclipse. This problem was fixed not by modification of the Curry IDE, but by modifying the implementation of the analysis server, which can now handle an arbitrary number requests in the same TCP session.

The next fix was a minor one: The identifier describing the extension point for visualizations contained spaces, which was not valid, because the identifier is used as an *element name* in an XML schema definition, and XML elements must not contain spaces [web:w3xml]. Removing the whitespace fixed this problem and finally it is possible to add implementations of the *visualization* extension point.

One simple visualization was added to the IDE. This visualization uses a text-box containing the string(s) returned from the analysis server as the output. This implementation was connected to the *text* format provided by CASS. The other two formats, *CurryTerm* and *XML* have no visualization yet. If the user chooses these other formats, the output will remain empty (except for the "success" message noting that the analysis request has finished successfully).

The current implementation is still far from perfect, we can identify multiple defects:

- ▷ The output is logically structured only by the requests; There is no option to show all analysis results for a specific function or to update older requests. There would be multiple options for increasing the flexibility of the user interface.
- ▷ It is bad that the user can access output types that do not actually have an visualization.

▷ When choosing the output type for a visualization, the options are grouped by the format used to transfer the information from CASS to the visualization. This format is irrelevant to the user.

For the latter points, we would make one suggestion: The interface for visualizations should be modified so that the visualization specifies its input format. Previously, the user would choose in this order:

1. the type of analysis to perform,
2. the type of format (*text*, *CurryTerm* or *XML*),
3. the specific visualization.

With the modified interface, it would be possible to simplify this process to choosing

1. the type of analysis to perform,
2. a visualization

where the visualization would internally uses one of the formats.

Two more points are noteworthy. Firstly, the changes to the code structure (or rather: the project structure) affect the analysis code. The major difference is that there is no separate project containing the interface and the extension point anymore. These changes are explained in more detail in Section 4.6.1. Secondly, the analysis system is now also used to implement type checking: For this purpose, a new type of analysis was added, called "TypePrinter". This new analysis simply prints the types of the functions defined at top level in the module. It is used to retrieve the typing information. For the details, see Section 4.4.

4.3 The Curry Console and Error Marking

The relevant packages are

▷ de.kiel.uni.informatik.ps.curry.uinonxtext.launch

▷ de.kiel.uni.informatik.ps.curry.uinonxtext.curryconsole

The Curry console is opened by launching the current project. When the user launches the project, the basic internal control flow is as follows:

▻ the *launch shortcut*[2] determines the *launch configuration* to use[3] and starts the *launch delegate*.

▻ the *launch delegate*[4] uses the information from the *launch configuration* to build the complete command and executes it in a separate process. The environment variables for this process are set according to both the system defaults and the project setting for additional library paths. This process is handed over to the *CurryConsole* class that handles the corresponding in- and output.

▻ the *CurryConsole* class handles both the in- and output between external process and the user, and the extraction of errors.

The last step (the *CurryConsole* class) is the most interesting, as it implements the extraction of errors from the output. It should be noted that this class does not implement the view for the user-interface; for that purpose, the generic console plugin provided in Eclipse is used. The interface provided by both the external process and the console plugin are streams. Consequently, if extracting the errors was not required, it would be sufficient to start threads that simply forward data from one stream to the other.

In order to implement recognition of errors, the output stream from the external process needs to be parsed. Effectively, there are now two consumers for the data in that stream: The output to the user (via the console plugin) and the extractor that parses the data and reacts to error messages. For this purpose, a thread was implemented[5] that forwards a stream to multiple destinations. This splitting is implemented with one specific property: The data is gathered (cached) and forwarded in chunks, based on a timeout. To understand what exactly happens and why it is necessary, we need to consider the nature of the external process.

The interpreter started in the console is an *interactive* session, i.e. the user can repeatedly enter expressions to be evaluated[6]. While this is pleasant to the user, it has one consequence when reading the output: The program does not end after one command, so there is no defined end of the output, and one can not directly tell when the output of one command ends and the output of the next starts. If there are errors, the complete output always terminates with the string

[2]launch.CurryConsoleLaunchShortcut

[3]if the user selects a specific *launch configuration*, this step can be omitted

[4]launch.CurryConsoleLaunchDelegate

[5]curryconsole.StreamTimeoutChunkedSplitterThread

[6]or executed, in the case of IO-actions

```
ERROR occurred during parsing!
```

But that does not help as there might not be any error, so waiting for that specific string is not an option. Fortunately, there is a simple time-based heuristics that seems to work perfectly fine: Consider all output that happens in the same time-frame as one block, and break blocks when there is no new output during a certain (small) amount of time[7].

These chunks are then processed in a custom output-stream that parses each chunk and adds the necessary markers to the sources[8]. For the parsing, a regular expression is used to retrieve the file name, the line number, and the message.

I have described how markers are added, but it is necessary to remove them as well, when the corresponding error no longer applies. The interpreter prints a message for each module it compiles, regardless of whether any errors occur. This allows a straightforward implementation: Whenever a Curry module is compiled, all markers for that module are deleted. After that, the new errors (if any) are considered and new markers are created.

[7]the value used in the source code is 10 milliseconds - too small to be noticeable to the user
[8]curryconsole.PakcsParserChunkOutputStream and curryconsole.PakcsOutputParser

4.4 Type Checking

When the user invokes the compiler for a Curry program, the compilation might fail with one or more type errors. Because these errors can happen rather often, it is the aim of this implementation to provide faster feedback about typing errors to the programmer. One obvious approach would be to automatically call the compiler regularly in the background. While this would certainly satisfy all our needs (it would catch all compile-time problems, not only type errors), there are two reasons to re-implement type-inference / type-checking to some degree: On the one hand, calling the compiler every few seconds (continues checking would result in such a rate while editing) could cause noticeable CPU load. But more importantly, other features in the IDE require more detailed type information than the existing external tools can provide at the moment[9].

However, by using the analysis framework (see Section 4.2) it is possible to retrieve type information of all top-level functions. This means that we can simplify the (re)implementation of the type inference algorithm: In module-scope, we can use the externally provided information and just *check* the types. Local to functions, the full type *inference* is implemented, based on inference algorithm as described in [Damas/Milner 82]. One thing to note is that user-given type signatures for a function provide global type information without the need to call the analysis framework. However, we do not want to depend solely on the user's signatures for our functionality.

The external analysis framework has one notable property: It does only work if its input (the source code) compiles without error. Without further tricks we would have no access to global type information exactly in the cases where there are any errors that we want to detect. To solve the problem, the typing information needs to be cached. Which raises the question of how to operate the cache: Firstly, when do we try to refresh the information, i.e. when and how often do we call the analysis framework? Secondly, how long the information in the cache is valid? If the body of a function is modified, the type previously inferred might no longer apply, and keeping it in the cache could result in false reports of type errors. Our goal is to avoid any such false positives, so we need to invalidate certain parts of the cache.

[9]the observational debugger must know the type of the expression being observed

4.4.1 The Type Cache

It seems inefficient to call the analysis framework each time we execute a check (i.e. roughly every few seconds when editing) in the hopes of finding a snapshot where the program actually compiles. The probability of success is low, as a program rarely is valid in the middle of editing, and there is a computational cost to starting the analysis so frequently. There is one event that happens less frequently and which has a higher probability that the code compiles without error: When the user saves the source code in the IDE. This event[10] is used to trigger the refreshing of the global type information data.

The next question is when to remove type signatures from the type cache. When a method body changes, the type retrieved at the last update may no longer be correct. Let us look at one simple example:

```
foo = True
-- to be changed to
-- foo = ’a’

bar = foo
-- type depends on type of foo

baz = foo==bar
```

Now, we make three assumptions: Firstly, that the user saved the document and the type cache was updated using the external analysis tool. The cache contains (at least) the following type signatures:

```
foo :: Boolean
bar :: Boolean
baz :: Boolean
Prelude.(==) :: a -> a -> Boolean
```

[10]technically, the *build* event in Eclipse is used. But when the option *build automatically* is enabled, which it is by default, each save triggers a *build*.

Secondly, that the user modifies the body of `foo` as described in the source comment. And thirdly, that the user has not saved the document again, i.e. the cache has not been updated yet.

Note that the modified code is still type correct; the types of `foo` and `bar` have changed, but that does not cause any problem in `baz`. The local type inference would return `Char` as type of the body. If the cache still contained `foo :: Boolean`, the IDE would report exactly one (false) type error, as the type checking algorithm would make the assertion of the signature and the inferred type to be equal. There would be no type error in *baz*, because the type inference algorithm would only use the type signatures from the cache.

To eliminate the false error, the signature for the changed function, `foo`, is removed, unless the signature is annotated in the source code. The type check would no longer report an error for `foo`, because there is no signature to compare the inferred type against. And for both `bar` and `baz`, this change does not create problems: If an global variable has no entry in the type cache, its type is supposed to be unknown, and treated as arbitrary.

The cache, however, does contain one outdated piece of information: the signature for `bar`. While this does not cause any problem in given code, it is easy to imagine a modification to `baz` where it does cause a problem:

```
baz = bar=='b'
```

Remember that the type cache still contains the signature `bar :: Boolean`, because the body of `bar` never got modified. Consequently, the type inference algorithm reports a type error, because the type of `(==)` necessitates that the type of `bar` be equal to `Char`. To avoid this kind of error, more type signatures would need to be deleted on any modification. One strategy would be to recursively delete the signatures[11] of all functions that contain a reference to the modified function from the type cache. While such a "brutal" method would indeed eliminate any false errors, it might often eliminate *any* errors: The recursive deletion might include large parts of the program, effectively clearing the type cache and making most of the type checking useless. For this reason, the implementation does not delete recursively, even if there are certain special cases where false errors will be reported. If the user suspects that the IDE reports a false error, he/she can save the program and trigger an update of the type cache, so that any false information is replaced.

[11]again, if any signature is provided by the user, no deletion or further recursion is necessary, but we focus on the "worst" case that there are no signatures provided in the source.

4.4.2 Type Cache Data Structure

Generally, a workspace can contain more than one project. Because different projects do not share a namespace, there must be separate type caches in order to prevent name clashes. Therefore the data structure is a map from project identifier to project-specific type cache data. This project-specific cache is a map from qualified Curry identifier (i.e. the function) to a structure containing:

▻ a type representation,

▻ a hash of the relevant source code

To detect whether a function was modified, a hash of the source code section corresponding to the function is used. When a single function is changed, the offset of more than one function in the same file might change, but the hashes of other functions are not affected.

To understand what is meant by the phrase *relevant* in the above definition, let us look at an example: Figure 4.1 contains a the complete source belonging to one function. In this case, a type signature is present which annotates the type of the function, regardless of what we might infer for the function (in this example, there is a type error, but the signature does not care). Because changes to the body are irrelevant for the cache, the *relevant* source code is just the type signature. Therefore, fixing the type error in the equation 1 does not affect the cache.

```
fib :: Int -> Int             -- signature
fib 0 = '1'                   -- equation 1, containing minor mistake
fib 1 = 1                     -- equation 2
fib n = fib (n-1) + fib (n-2) -- equation 3
```

Figure 4.1. Grammatically, a function can have more than one relevant node in the syntax tree: In this example, it has four. Because a signature is present, we want the type `Int -> Int` to be used in the cache, ignoring the type error in the first equation. Unless the signature is changed, the type does not change.

On the other hand, if no type signature was present, any equation can have an influence on the type. See Figure 4.2 for an example. Consequently, here the *relevant* source code is the hash over all equations of the function. This way, any change to any equation is considered a potential change of the type.

Modifications to the type cache of a project are always done per-module. To go below that, i.e. to make updates per-function, is hard to implement: A function can comprise of more than one node in the syntax tree; also, the signature does not even need to be placed close to the respective equations. Because hashes are used to detect changes, this decision is not inefficient either.

```
foo 0 = (+0)    -- this equation could be omitted, but
                -- it would change the type of the function.
foo n = id
```

Figure 4.2. `Int -> Int -> Int` or `a -> b -> b`?

The type cache is saved persistently, so when the program starts, the cache can be loaded and does not need to be updated by calling the external analysis tool. If a project is in a state where it does not compile successfully, any type errors are preserved even after restarting the IDE. The persistent data is currently written out every time the cache is updated successfully (and read when Eclipse is started, of course).

4.4.3 Control Flow and Code Structure of the Type Checking

Altogether, the implementation of the type inference/type checking mechanism involves six packages in two projects (*CurryIDE* and *CurryIDE.uinonxtext*). This distribution is caused by the different ways the classes are being accessed; for example the cache is written on a build event, while it is read in the type inference algorithm every few seconds. Whilst the build event is a user interface event, the type inference algorithm does not belong to the user interface code. Furthermore, the structure is determined by the dependency restrictions between the projects: It is not possible to access the *ui* project from the *CurryIDE* project. The resulting groups are:

1. types/* in CurryIDE
2. type-check/* in CurryIDE
3. typing/TypeCache.scala in CurryIDE.uinonxtext
4. BuildCheck.java in CurryIDE.uinonxtext

5. validation/CurryJavaValidator.java in CurryIDE
6. typeinferrer/* in CurryIDE.uinonxtext

item 1 contains the classes for representing Curry types and some basic utilities, including a parser for types.

The type-inference/checking algorithm and some extra types needed for the algorithm are defined in item 2; it also contains the core cache data structure.

TypeCache.scala (item 3) manages some aspect of persistence for the type cache. It provides methods to load and store the data and automatically loads the data at start-up, using the *IStartup* extension point in Eclipse.

To fill the cache, *BuildCheck.java* (item 4) implements the *IXtextBuilderParticipant* extension point provided by Xtext. Whenever a build is executed, the *build* method is executed. For each module modified since the last build, the external Curry analysis is used to retrieve the types of all top-level functions (assuming there are no errors in the source). The cache is then filled with that data.

The *CurryJavaValidator* (item 5) is provided by Xtext to allow the plugin developer to add custom validations to the IDE. These checks are executed whenever a Curry source file modified and it is possible to add error markers when any problems are detected. Two validations are used to implement type checking:

▷ method `dropDirtyTypeCacheInfo`: Whenever a module is modified, the types of top-level functions that are removed from the cache (using hashes as described in the previous chapter).

▷ method `checkFunctionType`: For each modified top-level equation, the type inference algorithm (item 2) is executed (using the type cache for the respective project) to check for type errors. If an error is found, a new error marker is created. Markers are automatically deleted when a new check returns without any problems.

The two classes in (item 6) implement the typing functionality for the user, and provide the respective UI-elements. When the user selects an expression and uses the corresponding context menu item, the `execute` method of the *ExpressionTypeInferrer* class is executed. It uses the type inference algorithm (item 2), applied to the surrounding function, to determine the type of the selected expression. This type is displayed to the user using the *InferredTypeView*.

4.4.4 The Type Inference Implementation

Our description will be divided into two parts: The first part will describe the general process of the type inference. The second part will focus on type error messages: With certain small modifications to the algorithm described in the first part we can add additional information to the messages returned in the case of type errors. An example for additional information is positioning, such as "at Main.curry:22.13".

The implementation is based on the algorithm as described in [Han13, p. 51-55]. Its aim is to derive a (most general) type for the function (and of all its sub-expressions) so that the result is type correct. As input, the algorithm uses the syntax tree of a single function and the type signatures of any top level functions used in that function. The output is either an error or a mapping from all (sub)expressions in the respective function to types. The referenced algorithm applies to simplified language with the grammar (EBNF):

```
function = identifier pattern* ['|' expr] '=' expr
pattern  = identifier
         | '(' identifier pattern ')'
expr     = identifier
         | '(' expr expr ')'
         | '(' 'if' expr 'then' expr 'else' expr ')'
         | '(' expr infixOp expr ')'
         | '(' λidentifier → expr ')'
```

where `infixOp` is some rule for infix operators and `identifier` is a non-terminal for identifiers. The algorithm then consists of the following steps:

1. Create initial expression/type pairs:

   For a function $l\ p_0 \ldots p_n \mid c = r$ create the pairs

$$\begin{aligned} l\ p_0 \ldots p_n &:: a \\ r &:: a \\ c &:: \texttt{BoolOrSuccess} \end{aligned}$$

   and omit the last pair if there is no guard, i.e. for $l = r$. We use a special type `BoolOrSuccess` internally to express the cases were both `Bool` and `Success` would be valid.

2. Transform the expression/type pairs: Replace

   ▷ $(e_1\ e_2) :: \tau$ by $e_1 :: a \to \tau,\ e_2 :: a$ (where a is a new type variable),

   ▷ $($`if` e_1 `then` e_2 `else` $e_3) :: \tau$ by $e_1 ::$ `Bool`, $e_2 :: \tau,\ e_3 :: \tau,$

   ▷ $(e_1 \circ e_2) :: \tau$ by $e_1 :: a, e_2 :: b, o :: a \to b \to \tau$ (a, b new type variables) (where $\circ$ is an infix operator)

   ▷ $(\backslash x \to e) :: \tau$ by $x :: a, e :: b, \underbrace{\tau \doteq a \to b}_{\textit{type equation}}$ (a, b new type variables)

   ▷ $f :: \tau$ by $\tau \doteq \sigma(\tau')$
   if f has a known type $\forall a_1 \ldots a_n : \tau'$ and
   $$\sigma = \{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$$
   where $b_1, \ldots b_n$ are new type variables.

   The result of the transformation contains both a set of expression/type pairs and a set of type equations.

3. Transform pairs to type equations:

   In the result of the last step, all pairs are of the form $x :: \tau$ (where x is an identifier). For any two pairs $x :: \tau_1$ and $x :: \tau_2$ create the type equation $\tau_1 \doteq \tau_2$.

   For the next step, the input is the union of the type equations created in the last two steps.

4. Solve the system of type equations, i.e. find a "most general unifier" σ, i.e. a mapping from type variables to types. We define:

   ▷ A substitution σ is called *unifier* for a system of equations E if $\forall l \doteq r \in E :$ $\sigma(l) = \sigma(r)$. We apply σ on type expressions by canonical expansion.

   ▷ A substitution σ is called *most general unifier* (mgu) for a system of equations E if for any unifier σ' for E there exists a substitution φ with $\sigma' = \varphi \circ \sigma$ (where $(\varphi \circ \sigma)\,(\tau) = \varphi(\sigma(\tau))$)

   To find the mgu, we apply the following transformations rules, as described in [MM82], to our system of type equations:

Decomposition:	$\dfrac{\{k\,s_1 \dots s_n \doteq k\,t_1 \dots t_n\} \cup E}{\{s_1 \doteq t_1, \dots, s_n \doteq t_n\} \cup E}$	k type constructor
Clash:	$\dfrac{\{k\,s_1 \dots s_n \doteq k'\,t_1 \dots t_m\} \cup E}{\texttt{fail}}$	$k \neq k'$ type constructors
Elimination:	$\dfrac{\{x \doteq x\} \cup E}{E}$	x type variable
Swap:	$\dfrac{\{k\,t_1 \dots t_n \doteq x\} \cup E}{\{x \doteq k\,t_1 \dots t_n\} \cup E}$	x type variable
Replace:	$\dfrac{\{x \doteq \tau\} \cup E}{\{x \doteq \tau\} \cup \sigma(E)}$	x type variable, occurring in E but not in τ, $\sigma = \{x \mapsto \tau\}$
Occur check:	$\dfrac{\{x \doteq \tau\} \cup E}{\texttt{fail}}$	x type variable, $x \neq \tau$, x occurring in τ

If no more rules can be applied, the result is either `fail` or a set of equations which represents an mgu.

Let us retrace these steps in a simple example: We consider the function

```
f = \x -> x + 1
```

1.

$$\begin{aligned} f &:: a \\ \lambda x -> x + 1 &:: a \end{aligned}$$

2.

$$\begin{array}{rcr} \lambda x -> x + 1 :: a & \Rightarrow & x :: b \\ & & x + 1 :: c \\ & & a \doteq b \rightarrow c \\ x + 1 :: c & \Rightarrow & x :: d \\ & & 1 :: e \end{array}$$

$$\begin{array}{lcr} & & + :: d \rightarrow e \rightarrow c \\ + :: d \rightarrow e \rightarrow c & \Rightarrow & d \rightarrow e \rightarrow c \doteq \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int} \\ 1 :: e & \Rightarrow & e \doteq \mathit{Int} \end{array}$$

We can visualize the expression/type pairs: $\overbrace{f}^{a} = \overbrace{\lambda \underbrace{x}_{b} \rightarrow \underbrace{\underbrace{x}_{d} + \underbrace{1}_{e}}_{c}}^{a}$

3. The final pairs are:

$$f :: a$$
$$x :: b$$
$$x :: d$$

resulting in one more equation: $b \doteq d$.

4. The mgu is calculated for
$\{a \doteq b \rightarrow c, d \rightarrow e \rightarrow c \doteq \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}, b \doteq d\}$.

It returns (successfully) with the mgu
$\{d \doteq \texttt{Int}, e \doteq \texttt{Int}, c \doteq \texttt{Int}, b \doteq \texttt{Int}, a \doteq \texttt{Int} \rightarrow \texttt{Int}\}$.

This algorithm for the simplified language can be extended to work with all the syntactical features of Curry. The corresponding modifications only affect steps 1 and 2. Because Curry has a more complex grammar for expressions, the number of cases to differentiate in step 2 is rather large and we will not formally describe the process for each case here, as they do not contain anything fundamentally new.

Still, we would like to highlight a few key differences between the abstract algorithm and the actual implementation:

▻ The input consists of the syntax tree for a function plus all known types of top level functions. References in the syntax tree (e.g. from the use of a variable to its introduction) are already resolved, which simplifies the introduction of syntactical features that allow shadowing, i.e. the use of the same variable identifier in different scopes.

- ▻ When creating new type variables, the mapping from expression to type variable is saved separately. This mapping can be used together with the mgu to determine the actual derived type of any part of an expression. This is used when the user requests the type of an arbitrary expression.
- ▻ The steps one through three of the algorithm are handled by a single recursive descent into the syntax tree. Step two essentially is a recursive procedure on its own, and the other two steps can be executed "on the fly": We can introduce new type variables for sub-expressions as soon as we first refer to them, and create the respective type equations as soon as both sides of the equation are known.

Three classes[12] form the core of the inference implementation. The `FunctionTypeCheck` class contains the code to create the type equations, i.e. the methods that recursively descent into the syntax tree, dispatching over the different types of nodes encountered. Type equations are added to the `EquationBuilder`. When the set of equations is complete, the `EquationSolver` computes the mgu. The `FunctionTypeCheck` class contains two public methods: One with the purpose of checking the equations (used when validating the modules), and one for retrieving the type of a specific expression (used at the user's request).

To motivate the changes to the implementation as described so far, we look at the output in the case that the inference fails: It returns no information about what failed and where. Trivial modifications could add minimal information that could be expressed like:

```
type error: clash between Int and [] in the equation Int==[Char]
```

Such a message would be better than just "type error!", but it might still be rather cryptic to the user, as there is no information about the origin of that type equation or any positioning of the relevant types in the source code. In order to actually help the user in the case of an error, we need to modify the algorithm so that we can create proper error messages. The implementation adds three types of information to enrich the messages:

1. Positioning of the relevant sections in the source code,

[12]all classes can be found in de.kiel.uni.informatik.ps.curry.typecheck

2. The type constructors "above" the type that caused the error,
3. The *direction*: the distinction between what type was expected and what type was found.

To illustrate point number two, let us consider an example error message[13]:

```
Type error: expected [Char] but found [Int] at Main.curry:23:9
```

Without the surrounding type, the error would be `expected Char, found Int`. That is, for any type constructed with a type constructor, the algorithm prints the complete type, even if internally the error occurred for some sub-type. Note that there is an exception for the function type, i.e. the type constructor `(->)`. This strategy (i.e. to include any surrounding constructors other than `(->)`) seemed to produce reasonable results that neither were too small (like `Char vs Int`) nor too broad, like if the complete type signature of the function was used in the error.

In the basic algorithm, a type equation consisted of two type expressions. A type expression is either a type variable or a type constructor with a (possibly empty) list of type expressions as parameters (See Figure 4.3). You can think of a type expression as a tree, where each non-leaf node is some constructor, and each leaf node is either a type variable or a nullary type constructor.

EBNF for type expressions:

```
typeexpr ::= typevar
           | "(" typecons typeexpr* ")" ;
```

where `typevar` describes type variables, and `typecons` contains (→), the list type constructor `[]`, tuple type constructors (`()`, `(,)` . . .) and any custom type constructors like `Boolean` or `Maybe`.

Figure 4.3. a formal description of simple type expressions

In our implementation, we need additional data, and we add it at each node in this tree. This additional information consists of three values:

- ▻ an (optional) position, describing the location of the expression linked to this type expression in the source code.

[13]This message was created as an illustration; in the IDE, the error is shown slightly differently. Inter alia, the position is conveyed by the placing of the error marker, not in the message itself.

- ▷ a optional "context-type", referencing a "surrounding" expression. For example, the type variable v_1 might link to $[v_1]$. This information is used to print "[Char] is not [Integer]" instead of "Char is not Integer" in the error messages.
- ▷ a three-value flag called "expectedness" that is used for the *direction*, i.e. for making the distinction expected-vs-found. The three values are *expected*, *neutral* or *found*.

These additions result in the modified EBNF shown in 4.4.

EBNF for enriched type expressions:

```
typeexpr ::= typevar                     ["at" position] ["in" typeexpr] ["+"|"-"]
          | "(" typecons typeexpr* ")" ["at" position] ["in" typeexpr] ["+"|"-"] ;
```

where we use + for *expected* types and - for *found* types, and position is some representation of a position in the source code (e.g. line, column and file name). The other non-terminals are used as above.

Figure 4.4. a formal description of enriched type expressions

For a better understanding of the semantics of these members, we will start at one possible end of the process: The creation of the type error messages. There are two transformation rules that can result in a failure. A *clash* is the first possibility, the second is a failed *occurs check*. The latter is the simpler case; we print the two types and, if available, we mention the position of one of the sides in the output.

For a type clash, the first step is to inspect the context-types for both sides of the relevant type equation. Taking into account the way the context-types are constructed, we know that either both sides have context-types, or both have none. We also know that no context-type has a context-type. If present, the respective context-types replace the types at both sides in the rest of the procedure, otherwise, the sides remain unchanged. Because context-types do not have context-types, no recursion is necessary. Next, we consider the position values of both sides. If neither has a position value, the output will just use the position of the function equation. If exactly one side has a position value, this one is used in the output. If both have a position value, we use the innermost (i.e. more specific), rightmost position. As the last step, we compare the *expectedness* flags of both sides. Let us call the "more expected" type expression t^+, the "more found" expression t^-. Then, the error message will be "Expected t^+, found t^-". If the expectedness is equal, the output will just be "t_1 is not t_2".

For the function

```
foo = sum "abc"    -- sum :: [Int] -> Int
```

At some point during the calculation of the mgu, the type equation

$$\overbrace{\texttt{[Int]}+}^{\tau_1} \doteq \overbrace{\texttt{[Char] at } p_1}^{\tau_2} \tag{4.4.1}$$

is encountered. This equation will subsequently be de-composited into the equation

$$\texttt{Int in } \tau_1 \doteq \texttt{Char in } \tau_2 \tag{4.4.2}$$

which will cause the actual fail. When creating the error message, the first step is to check for context-type, so we switch to the equation $\tau_1 \doteq \tau_2$, i.e. to (Equation 4.4.1). The position p_1 will be used for in the output. Then, the expectedness is compared, creating the final message along the lines of
"at p_1: expected `[Int]`, but found `[Char]`".

So how do we arrive at either *expected*, *neutral* or *found*? When constructing the set of type equations, we define initial values, which may be modified during the calculation of the mgu. It might be noted at this point that only two transformations rules during the calculation of the mgu add new equations: Decomposition and replace. For swap, we can swap the complete enriched type expressions.

For the position member, the initial value for any type variable is the position of the corresponding (syntactic) expression. For the nodes in the type expression that are not type variables, we do not set a position, but this does not matter as there will always be equations binding these expressions to single type variables (which have a position). When a type variable is replaced by a different expression in the *replace* step of the mgu calculation, the positioning of the replacement is set to the position of the replaced variable, unless it was set already. The context-type is initially not set; it will be set during the decomposition step of the calculation of the mgu.

In the example above, this happens between (Equation 4.4.1) and (Equation 4.4.2). The decomposition changes the nodes of the type expression:

`Int` $\Rightarrow$ `Int in` τ_1 and `Char` $\Rightarrow$ `Char in` τ_2

As discussed before, the context-type is not set when decompositing type arrows. The only thing to consider is that we do not want context-types to have context-types. We enforce this property by making the transitive step before we assign, i.e. if type b has context-type c, and we want to assign b as the context-type (of some type a), we assign c instead.

The default value for the *expectedness* flag is *neutral*. Only in three cases it is set differently:

- ▻ At top level, when creating the type equation connecting the left hand and the right hand sides of an equation, set the variable for the right hand side expression to *found*,
- ▻ For any function type $a \to b$ created when compiling the type equations, set the parameter (in this case, a) to *found*,
- ▻ Set any `Bool` type created (and not “imported” as a part of a known type signature), e.g. for the condition in `if ... then ... else ...`, to *expected*.

The reasoning behind these rules will be explained later, but first we would like to give an example and show how the *expectedness* is transformed.

We consider the function

```
foo :: Int
foo = 'a'  -- can you spot the type error? ...
```

We are only interested in the *expectedness* flag, so we annotate only that one in the next type expressions. We write t^+ for a *expected* type expression and t^- for a *found* one. The equations created for `foo` would be:

$a = \text{Int}$ $\qquad$ a is a type variable connected to foo

$a = b^-$ b is the type variable for the right hand side of the equation
$b = \text{Char}$

We could see this as a chain, i.e. $\text{Int} = a$, $a = b^-$, $b = \text{Char}$.

The idea is that the expectedness flag provides a *direction* over this chain; in this case the right hand side of one of the equations is *found* and consequently the *direction* is "left is more expected". When transforming the type equations, the *replace* steps will make these chains shorter and shorter. Now, we dictate one property for any transformations, particularly for the replace steps:

▷ Any transformation must retain the expectedness-direction over any (implicitly given) chain in the set of type equations

Following this rule, for the example the result would be either $\text{Int}^+ = \text{Char}$ or $\text{Int} = \text{Char}^-$. In both cases the error message is "Expected Int, but found Char" (because `Int` is the more expected type) and this is the error message we want.

In the implementation, the only step being affected by this rule is the *replace* step, as it recalculates the flag for newly created nodes. We will not explain the exact algorithm that calculates the new *expectedness* flag, for two reasons: It would require a large case discrimination (what is the flag for the replacing variable? the replaced variable? the replacing expression?). Secondly, while we did not find counter-examples for our algorithm yet, the type-checking functionality is not even supposed to be perfect - it is intended to be a heuristics. What matters in the end is whether the Curry compiler detects type errors or not.

Note that internally, the implementation does *not* explicitly create chains. Also, chains do not need to be *decomposed* to be considered as such, i.e. we can imagine a chain $a = b$, $[b] = [c+]$, $c = d$ that connects a and d with a specific direction. This is important because we do no restrict the order of transformations.

Let us consider a second example where we can observe the effect of the *expectedness* flag inserted at arrow types:

```
foo :: Int -> Bool
foo = ...

bar = foo 'a'
```

When inferring the type for `bar`, the following equations are created:

$a = b^-$	a is connected to `bar`
	b is connected to the right hand side of the equation
$c = d^- \to b$	c is connected to `foo`
$c = \text{Int} \to \text{Bool}$	the type cache provides a signature for `foo`
$d = \text{Char}$	

After decomposing the type arrows, we can see a chain yet again: $\text{Int} = d^-$, $d = \text{Char}$, which is processed to either $\text{Int}^- = \text{Char}$ or to $\text{Int} = \text{Char}^+$. Either way, the error message would be "expected Int, but found Char". Note that the expectedness flag in the first equation does not have any effect in this example.

Now that we know how the *expectedness* is used and transformed, it is hopefully easier to understand the rules for setting the flag initially: Setting the flag defines the *expectedness* direction over any chains that involve the respective variable. Let us consider the rule for parameters in type arrows of type arrows more closely, using an application with annotated type bindings:

$\overbrace{\underbrace{g}_{b}\ \underbrace{x}_{c}}^{a}$ where g and x are arbitrary sub-expressions.

For this, we would create the equation $b = c^- \to a$, i.e. the type variable for the *parameter* is set to *found*. For both c and b, the algorithm will create certain other equations. Lets assume that simplifications result in equations $b = \tau_1 \to \_$ and $c = \tau_2$. These equations form a chain between τ_1 and τ_2, and the *found* flag we introduced will define the direction: τ_1 is the more expected one. This is what we want, because τ_2 represents the type inferred for the actual parameter, while τ_1 is the type expected by inspecting the type of *g*. The same line of reasoning can be used to construct the other two rules.

There is one case where this implementation seems to not produce the desired results:

```
foo :: a -> a -> a
foo = ...

bar = foo 42 'x'
```

The type signature of `foo` stipulates that the two parameters must have the same type. One might hope for an error message along the lines of "Expected Int, but found Char, at 'x'" (or vice versa for 42). Yet, this implementation will only print "Int is not Char". The reason is that the both expressions relevant for the error have the same level in respect to the structure of the type equations. The chain of type equations between `Int` and `Char` would contain two *found* flags, on different sides of equations. As a result, the total expectedness relation is *neutral*. This could be seen as a deficit of the implementation, but it is sensible: We can not really tell whether the first parameter or the second has the wrong type. A suggestion would be to have a new type of error message for this case, containing *both* positions and types in the error message. However, this is incompatible with the way Eclipse handles error markers (which have exactly one position), so this suggestion could not be implemented.

Appendix G is a Curry module containing a set of functions featuring different types of type errors. This file can be used to test the messages produced in the different cases.

4.5 Debugging

In order to explain what the implementation of the debugging features provided in the Curry IDE does, we would like to start with the user's perspective, and contrast the workflow of debugging with and without the IDE's features.

Not using the features, the programmer would have to modify the source code in order to insert some debugging function, like `trace`, at some expression he is interested in. The trace method has one parameter (a string) that serves as identification. When the programmer executes the modified program and the relevant expression being evaluated, the trace function would print some kind of note, including the identifier, to the console. The programmer would have to look for this kind of output. If there was more than a few traces, he/she might even have to make some mental effort to connect the identifier to the correct expression (if there are multiple traces present).

Using the debugging features, this process is somewhat simplified for the programmer. Firstly, we separate the actual program (i.e. the source code) and the formal description of what to debug (i.e. a list of debugpoints). To add a trace, the programmer does not modify the source code, but instead selects the wanted expression in the editor and adds a new debugpoint for that expression. The next difference is that the user does not need to specify an identifier. Lets assume that like above, there is one expression being "traced". When the programmer executes a debugging launch in the IDE, and the respective expression is being evaluated, the IDE would add an item to the *Curry Trace Debugging* view. This item would be internally connected to the respective expression, and the user can just double click it to retrieve that information.

We can summarize the features in the following points:

- ▻ Automation:
  - ▻ Adding debugpoints is just a few clicks;
  - ▻ The IDE handles logical connection from debugging output to debugpoints;
  - ▻ (for observations:) The parameter describing the type of the observed expression is automatically generated.
- ▻ Separation of Concerns:
  - ▻ There is no need to modify the original source;

▻ The original program's output is not mixed with debugging output anymore.

The implementation had one other focus, still: extensibility. It is possible to add new types of debuggers using Eclipse's plugin extension interface; this interface will be described in Section 4.5.4

The shortest description of how the implementation works would be: It does the same the programmer (as described above), but it hides it from the user of the IDE. We have two issues to address before we can continue.

At least behind the scenes we *do* need to modify the code in order to insert commands like `trace` for every debugpoint. But we do want both to hide this from the user and be able to undo any changes made. The obvious solution is to make a copy, and modify only one version. For this, a hidden folder is used that will contain a complete but modified copy of the project (with all its files).

The second issue is that the commands like `trace` provided by the Curry library print their debugging output to the console. This is inconvenient as we do want that information in the IDE, i.e. we would have to parse the output in the IDE. To avoid that, the debugpoints use new versions of these commands that print to a TCP stream instead. In Eclipse, our plugin starts a TCP server that listens to incoming messages and dispatches them. For example in the case of `trace`, the output is added to the `Curry Trace Debugging` view.

With these basic issues being handled, we can describe the full internal process used for debugging.

4.5.1 Basic Process of the Debugging in the IDE

Debugpoints are the core of all debugging functionality. They are connected to expressions in the Curry module (instead of line numbers, like normal *breakpoints* would be). The user can add and modify debugpoints. Debugpoints have a type that specifies what kind of *expression debugger* is to be used. This is the point where the extensibility comes into play: Using Eclipse's extension interface, new types of expression debuggers can be added at a later time. Debugpoints internally also have an identifier that will be used to connect debugging output back to the respective debugpoint.

When the user starts a debugging run, the following steps are executed:

▻ All files in the directory of the project are copied to the hidden debugging copy folder. If an expression is connected to the debugpoint, it is not copied directly, but modified in order to insert some specific debugging code. The exact modification is controlled by the *expression debugger*. The debugpoint's identifier will serve as a parameter to the debugging code.

- A TCP server is started that accepts and handles incoming debugging messages.
- Like for a normal launch, a console window is opened in the IDE, containing a Curry REPL. The difference is the current directory: It is set to be the hidden copy folder instead of the project's root. This way, the compiler (and consequently, the user) will use the modified sources.
- Each message received by the TCP server starts with the identifier of the debugpoint. This way, the message can be forwarded to the debugpoint (and further, to the expression debugger) to handle the message (and in most cases, display the message to the user in some way).

Note that it is possible to query the current directory from the debugging console, e.g. by executing the command `!pwd`[14]. This breaks the illusion to the user, but it is not avoidable.

4.5.2 Additional Potential Issues and Design Decisions

The first question is how to make a modified copy of the source code. To the authors best knowledge, Xtext does not provide any method of printing a modified syntax tree to a file. Luckily, the syntax tree nodes contain the string in the original input corresponding to that node. Therefore we can traverse over the syntax tree, print the strings for all leaf nodes and obtain a copy of the original module. To modify specific expressions, we can insert custom strings into the output when the respective node in the syntax tree is reached during the traversal.

Another thing to note is that apart from the modifications to the expressions, we need to add the imports for the modules containing the debugging functions, e.g. the modified `trace` method that prints its output to the TCP stream. This is implemented just like the other modifications: When we reach the module node in the syntax tree, the respective imports are added. There is one slight deficit in the current implementation: The insertions add a new line containing a non-indented import statement. But actually, the layouting of Curry allows all statements in a

[14]the exclamation mark can be used to execute a system command; here the *pwd* (print working directory) command is used.

module to be indented. The problem is, that they all need to be at the same level, and the inserted import statement could break that rule. On the other hand, in all existing Curry modules, the elements of the module are not indented. Still, to be conforming with the official Curry report, this needs fixing.

In addition the to last point, the imported modules must be accessible to the compiler, i.e. the folder containing the respective module(s) must be added to the Curry library path that can be configured in the IDE. This must be done manually at the moment.

A last question is when to refresh the modified copy. Problems might occur in multiple scenarios:

- ▻ One debugging console is opened, and the user starts a second debugging console in parallel;
- ▻ While one debugging console is open, the source code is modified;
- ▻ While one debugging console is open, the user modifies the list of debugpoints.

Unfortunately, it is not possible to make a "life update" while a program is running. Therefore, this implementation simply restricts the refreshes: The debugging copy is never refreshed while any debugging console is opened, i.e. the user needs to close any debugging console before updates occur. There is still the possibility that the user deletes the debugpoints while a program is debugged with that debugpoint being active. To prevent problems, all debugpoint identifiers are globally unique (i.e. they are not reused in any way) and the implementation just drops debugging messages where the identifier is no longer valid.

4.5.3 Code Structure and Control Flow for Debugging

Four packages contain the source code for the implementation of the debugging features (all in the project `CurryIDE.uinonxtext` and in the package `de.kiel.uni.informatik.ps.curry.uinonxtext`):

1. `launch/CurryConsoleLaunchDelegate.scala`

   The delegate that handles launches in both the debugging mode and the normal mode. Some special logic is used to handle the debugging launch.

2. `debug/*`

   The main package for the debugging implementation.

3. `debug/iface/*`

   Contains the two classes of the interface for expression debuggers.

4. `debug/impl/*`

   Contains the implementations of the expression debugger interface which are provided already.

There are three entry points (control-flow-wise) into the debugging package: The first is the modification of the list of debugpoints, the second the debugging launch, and the third the TCP server handling the debugging messages.

The list of debugpoints is internally represented using the `DebugPoints` class. The `DebugPointView` is the UI element containing the list of debugpoints; it handles the respective user input. When adding new debugpoints by selecting an expression in the editor and using the context menu, the `AddTraceHandler` class is invoked.

When a debugging launch is executed, the `CurryConsoleLaunchDelegate` computes the necessary environment information (e.g. the command to execute and the path environment variable). Then, it forwards the request to the `CurryDebugCopy-Synchronizer` that checks if there are opened debugging consoles present. If there are, a dialog lets the user decide what exactly to do. In the default case, when there are no open debugging consoles, the debugging copy is refreshed using the `ProjectDebugCopyHandler`. Then, the console is opened.

The TCP server for debugging messages, implemented in `CurryDebugSocketServer`, is started with Eclipse, so there is no need to explicitly start it when launching in debugging mode. When it receives messages, it will extract the debugpoint identifier from the message and use the `ExpressionDebuggerRegistry` to map this identifier to the debugpoint. The debugpoint updates a hit-count and then forwards the message to its expression debugger implementation. What exactly happens with the message further on depends on the type of expression debugger (i.e. for *trace* the handling is different than for *traceValue*).

4.5.4 The Expression Debugger Interface

A debugpoint is connected to an expression. Generally, we want something to happen when this expression is evaluated at run-time. We can think of multiple different "something"s:

▷ *trace*: We want only to know *that* the expression was evaluated.

▷ *traceValue*: Like trace, but we also want to know what the expression evaluates to.

▷ *observe*: We want to follow the exact control flow of the evaluation of this expression and its parts.

But this list is certainly not complete. The expression debugger interface is introduced to allow for future additions to the list. For any behavior we want, e.g. for *trace*, *traceValue* and *observe*, we can define implementations of this interface. Using Eclipse's extension mechanism, these implementations can be added as separate projects (or even: as separate plugins). Any implementations present in the Eclipse installation will be available to the user when he/she selects the type of expression debugger to use for a debugpoint.

The Curry IDE currently has implementations for the *trace*, *traveValue* and *observe*, and one "null"-implementation that does nothing and is mostly supposed to serve as a minimal example of an implementation for developers.

The interface consists of two classes: `IExpressionDebuggerFactory` and `IExpression-Debugger`. We will refer to these with *factory* and *debugger*, respectively. The factory is used to obtain instances of the debuggers. When debugging, there is one instance of a debugger connected to each debugpoint. On the other hand, the factory is not connected to any specific debugpoint, but describes this type of expression debugger in general.

The two interfaces are:

The `getTypeName` method returns an identifier for each type of expression debugger. The method in the debugger is just a duplicate and must return the same value as the factory it was created with. When the user selects the type of expression debugger to use for a debugpoint, these identifiers will be used.

The `createExpressionDebugger` method gives the factory its name: This method is used to create debugger instances. These instances will be connected to specific debugpoints.

The debugging code inserted by any expression debugger may refer to custom debugging functions from its own modules. These modules must be imported in any module that contains the respective inserted function. To simplify things, we

IExpressionDebuggerFactory

```
getTypeName():                                      String
getNecessaryImports():                              Set<String>
createExpressionDebugger():                         IExpressionDebugger
debuggingRefreshNotification(project: IProject):    void
```

IExpressionDebugger

```
getTypeName():                                      String
printCode(debugExprNode:            INode,
          debugExprType:            Type,
          outputStream:             OutputStream,
          curryStreamWriterFunction: String,
          printer:                  CurryPrinter):  void
addOutput(s:                        String,
          debugPoint:               DebugPoint):    void
```

Figure 4.5. interface members

just add all imports everywhere, regardless of whether they are actually necessary. For this, each expression debugger factory specifies a list of required imports using the `getNecessaryImports` method. When making a debugging copy of the code, the imports of all available expression debugger factories will be collected and then inserted to all modules.

When the user stops one debugging run and starts a new one, we would like to enable some form of feedback to the user so he/she can separate the last run's debugging output from the current output. However, the output is handled by the expression debuggers, so there is no way of uniformly adding such a separation. Instead, the `debuggingRefreshNotification` is used to notify all the expression debugger factories when a new debugging run is started. The factory can then either directly add some feedback to the expression debugger's output or notify its debugger instances.

The two remaining methods in the debugger are `printCode` and `addOutput`. The latter is called for every debugging message the TCP server receives for this debugpoint. It is responsible to forward this information to the user in some way. The first parameter is the message (that was passed to the the output method in the inserted debugging code). The exact data flow of these messages will be described in more detail further below.

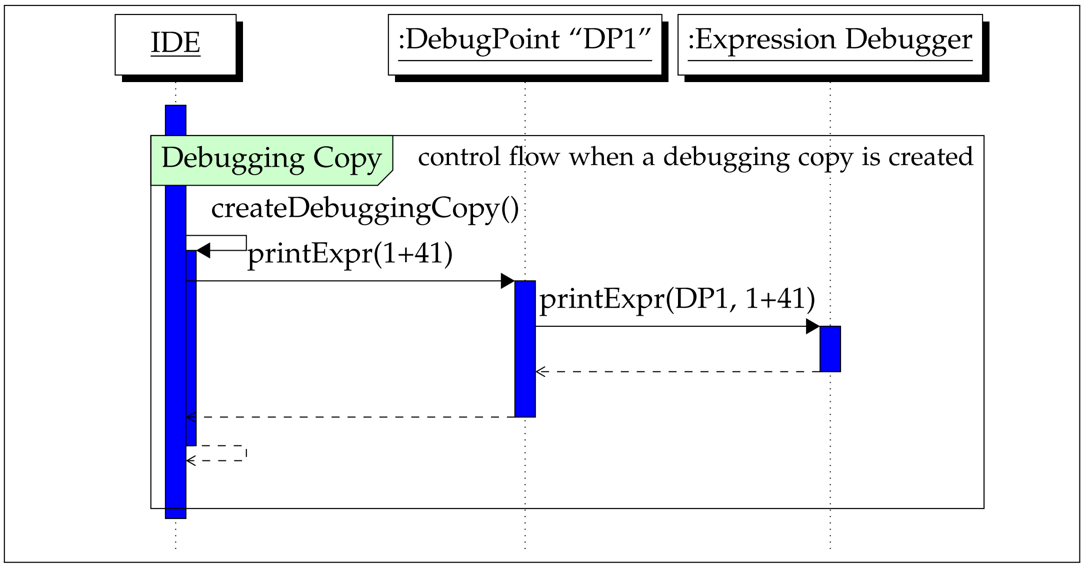


Figure 4.6. Sequence diagrams for the data flow when debugging using traces, part A

When creating the debugging copy of the source code, the expression debugger determines what debugging code to insert around the debugpoint's expression, using the `printCode` method. When the TCP server receives messages, they are first forwarded to the debugpoint and then to the expression debugger instance using the `addOutput` method. The semantics of the five parameters are not self-explanatory and will be explained below.

To understand the data flow of the debugging messages, let us devise the process at one simple example. Assume there is one debugpoint with identifier "Debug-Point1" and type *traceValue* connected to the expression `41+1`. As a first idea, when modifying the code, we could replace that expression by:

```
(unsafePerformIO $ do
  let x = (41+1)
  sendToEclipse "DebugPoint1" (print x)
  return x
)
```

The function `unsafePerformIO :: IO a -> a` is from the Curry library `Unsafe`; it performs an IO-Action outside of the IO-monad. It is considered unsafe, as it allows the programmer to embed side-effects into pure functions. We have to use it here, because sending a message to Eclipse is a side-effect (an IO-Action); and we

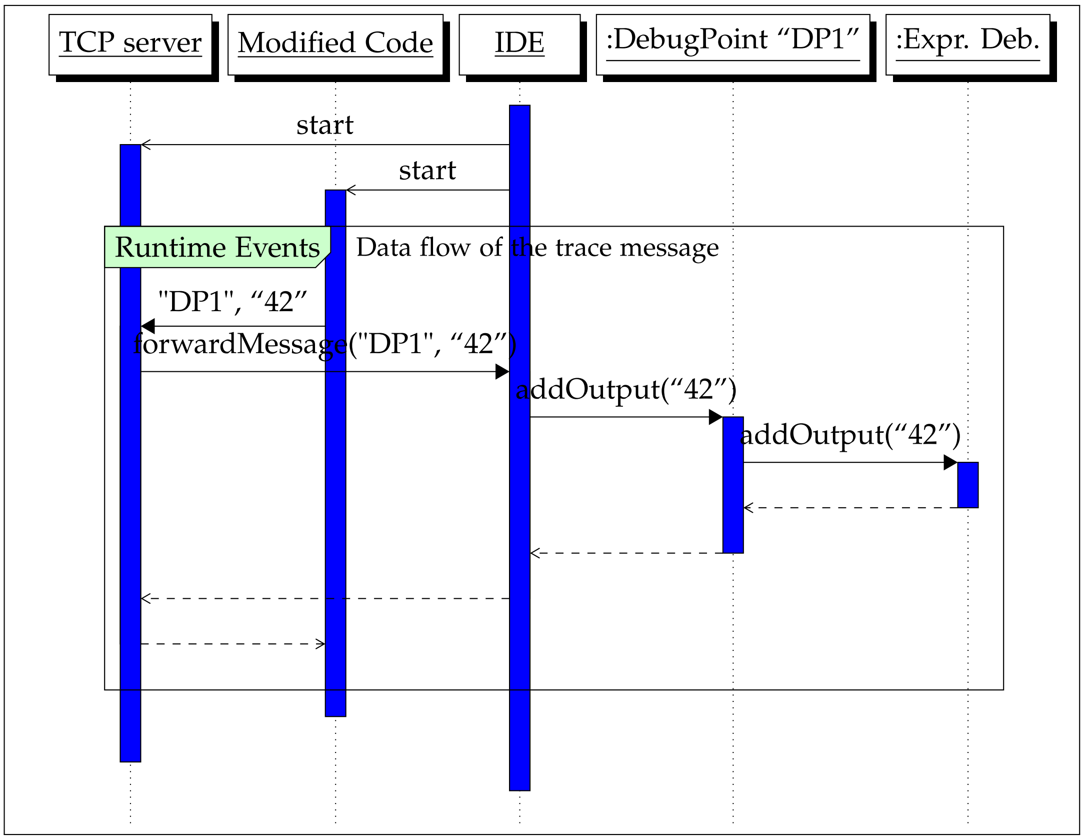


Figure 4.7. Sequence diagrams for the data flow when debugging using traces, part B

want to be able to insert this debugging code anywhere (i.e.: even in non-IO-code). `sendToEclipse` is some function that connects to the TCP server and transmits the debugpoint's identifier plus a message. The message would in this case contain the value of `41+1` after evaluation.

Note how we use parentheses both around the whole expression and the expression being traced. Because we insert additional code, the safe course of action is to insert these everywhere to prevent any problems (and the generated code will not be read by the user, either). In the following code snippets, some of the additional parentheses will be omitted for brevity.

Now consider what parts of this fragment are specific to this debugpoint: Only `"DebugPoint1"` and `41+1`. So in the `printCode` implementation for *trace* we might use a code template along the lines of:

```
unsafePerformIO $ do
  let x = EXPRESSION
  sendToEclipse IDENTIFIER (print x)
  return x
```

For this, we would have to pass the debugpoint's identifier to the expression debugger. Figures 4.6 and 4.7 show the control flow for both the code modification and the effect of the modification, i.e. the message forwarded back to the expression debugger when this specific code part will be executed.

We can, however, make two observations which will motivate one small change:

▻ The `sendToEclipse` method will always be called with the identifier as the first parameter.

▻ There is only one reasonable use for the identifier: As parameter of the `sendToEclipse` method.

And these statements are true independent of the type of expression debugger. This motivates the use of a slightly different template:

```
unsafePerformIO $ do
  let x = EXPRESSION
  SEND-MESSAGE-TO-SELF (print x)
  return x
```

That is, we simply encapsulate the partially applied `(sendToEclipse @IDENTIFIER@)` and provide *that* as a parameter to the `printCode` function. `SEND-MESSAGE-TO-SELF` has a semantic of "forward a message back to the expression debugger that printed it". This way, the expression debugger does not even need to know about the details of identifying messages. In the interface, this parameter is called `curryStreamWriterFunction` (the fourth parameter).

We have to be aware of one possibility: debugpoints can be nested. For example, we could also add a debugpoint just for `41` (in the expression `41+1` in the example above). When copying the code, inserting modifications, there is a function recursively descending into the syntax tree and calling the expression debugger's `printCode` function if the respective node is reached. Because of the possibility of nesting, we need the control flow to return from `printCode` back to this recursive process when we print the original expression in `printCode`. The sequence diagram in Figure 4.8 shows how the control flow jumps between the `CurryPrinter` (the class that manages the recursion) and two expression debuggers.

For the *traceValue* expression debugger, we considered the template for the replacement:

```
unsafePerformIO $ do
  let x = EXPRESSION
  SEND-MESSAGE-TO-SELF (print x)
  return x
```

Here, to print `EXPRESSION`, we need to return to the recursive process. This is the objective for the first and fifth parameter passed to `printCode`: The `CurryPrinter` class implements the recursive descent, and will be called with the `node` as parameter to print the expression.

Two parameters in the `printCode` method in the interface are not explained yet. `debugExprType` contains the type of this debugger's expression. It is currently only used by the *observe* expression debugger implementation, which needs the type to print corresponding code. `outputStream` is simply the output to be used - in the default case, it will be a file-output-stream for the copied version of the module containing the code being modified.

4.5.5 Expression Debugger Implementations

Internally, five expression debugger implementations exist, of which three are useful to the user. These three are `TraceDebugger`, `TraceValueDebugger` and `ObservationDebuggerExternal`. The `ZeroDebugger` is intended to be a minimal example for developers that provides no output at all (i.e. it leaves its expression semantically unmodified). The other unused implementation, `ObservationDebuggerInternal`, is unfinished. We will have a closer look at these implementations.

`TraceDebugger` and `TraceValueDebugger` are very similar and they share the same output view to display events. Both use the `EclipseDebugWriter` module that contains the implementation of what we called `sendToEclipse` in the source fragments above. Also, this module provides utility methods. Consequently, both implementations return the set {"EclipseDebugWriter"} from the `getNecessaryImports()` method.

The code for the `addOutput` method in `TraceValueDebugger` is

```
@Override
```

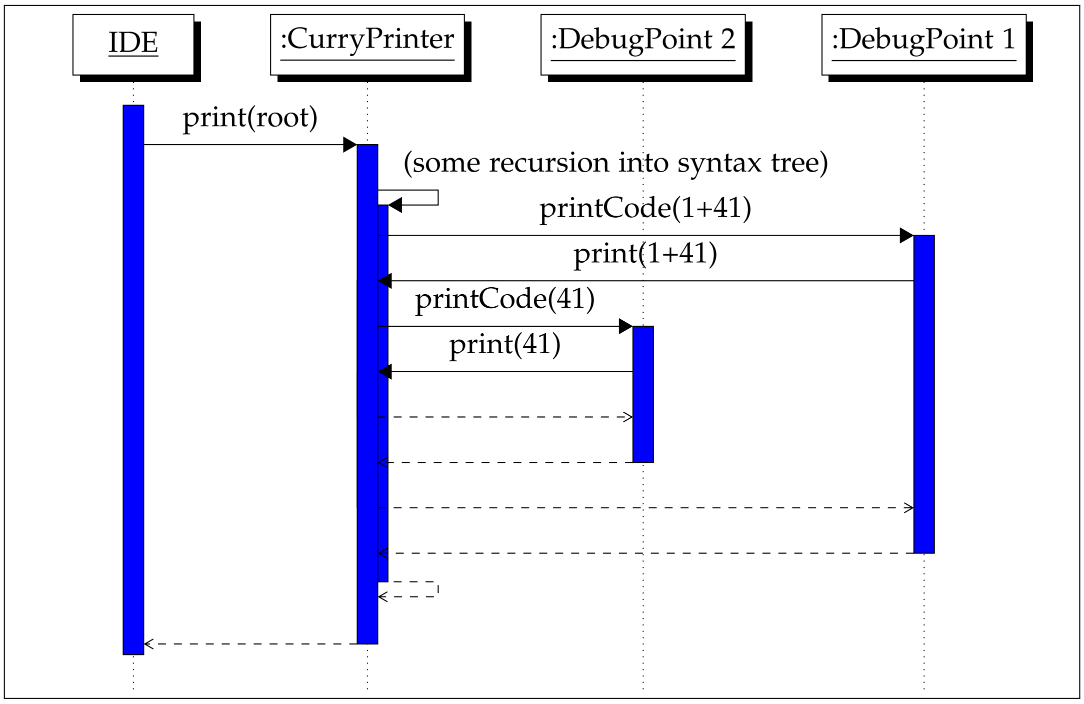


Figure 4.8. sequence diagram when printing the modified source code for debugging purposes, with two nested expression debuggers. The DebugPoints include the respective expressions debuggers here (hiding an additional indirection); splitting those two up would make the diagram unnecessarily complex.

```
public void addOutput(String s, Object debugpoint) {
  traceView.addLine((DebugPoint) debugpoint, s);
}
```

That is, we simply forward the output to the GUI element (`traceView`) that was created for this purpose. The view will show a list of the trace events, and the user can double-click the items in the list to highlight the respective expression in the source code. When a new debugging run is started, we simply add a placeholder line in this view to separate old and new events. This is implemented by calling the appropriate method in the `debuggingRefreshNotification()` implementation.

The body of the `printCode` method for the `TraceValueDebugger` is:

```
outputStream
    .write(("traceValue " + curryStreamWriterFunction + " (")
        .getBytes());
printer.printToStream(debugExprNode);
outputStream.write(")".getBytes());
```

First, we print the strings `"traceValue "`, `curryStreamWriterFunction` and `"("`. Then we call the printer to print the expression being traced, and then we add the closing parenthesis `")"`. Looking at the implementation of `traceValue`, the reader will be able to see that this essentially is equal to the implementation discussed above.

Finally, we would like to explain why there are two implementations (one of which unfinished) of the observation debugger. The difference between the two is the output for the user: For the *external* variant, the native (external) GUI provided by COOSy must be used to view the debugging output, while the *internal* variant was intended to have a viewer integrated into the UI of Eclipse. The unfinished variant contains modifications to the COOSy code that writes all output to the Eclipse TCP stream, but it lacks any interpretation of this output and proper visualization in Eclipse.

When using the *external* variant, the user must start the COOSy user interface by executing the command `:coosy` in the debugging console.

4.6 Overview of Changes

4.6.1 The Project's Structure

Xtext is the basis of the plugin, and repeating the description in Section 2.3.1, by default an Xtext implementation consists of four Eclipse projects: *foo*, *foo.sdk*, *foo.test*, and *foo.ui* (where "foo" is the project's name). *foo.sdk* does not contain any source code; it serves as a formal description that bundles the other Eclipse projects. *foo.test* should contain only test code, so the remaining *foo* and *foo.ui* would contain the true implementation of the plugin.

The work of our predecessor consisted of six Eclipse projects in total; the four mentioned above plus an *update site* and the *analysis* project. The update site is, like the *.sdk*, more of a formal necessity that allows the plugin to be installed in Eclipse. The *analysis* project does contain the description of the interface for visualizations of the results from the external CASS utility. Unfortunately, Marian Palkus did not provide a reason for creating the separate *analysis* project in his thesis.

The structure of the project was changed in the following ways:

1. A new project called `CurryIDE.uinonxtext` was created that contains features not connected to Xtext. All classes connected to Xtext remain in the `CurryIDE.ui` project. The are two reason for this separation. Firstly, the implementations needed by Xtext are semantically different from the additional features that where added to the IDE and *should* be separated, just like UI and non-UI features are separated. Secondly, Xtext establishes a rigid structure: It expects numerable classes to be defined in specific packages, and changing the structure did not seem possible (we tried, unsuccessfully, to find Xtext documentation about this topic, and simply refactoring did not work). Without the possibility to group packages, adding even more classes would lead to a confusing set up where it is hard to find individual classes.

2. In continuation of the last item, certain packages were internally restructured when moving them to `CurryIDE.uinonxtext`. The focus lay on grouping all classes by the general feature they implement, avoiding unnecessary sub-packages.

3. Previously, the analysis visualization extension point together with the corresponding interface were defined in its own project. This additional project was removed and incorporated into the `CurryIDE.uinonxtext` project. The reason for this integration is that the analysis visualization extension point is strongly

connected to analysis implementation, and there seemed to be no benefit from using a separate project.

I personally can see one option where (additional) new projects might make sense: For the implementations of both extension points provided in the CurryIDE. These two are the analysis visualization and the expression debugger extension points. For both, the IDE currently contains at least one implementation, and these implementations currently reside in `CurryIDE.uinonxtext`. However, as the implementations could be omitted without causing problems, separate projects might be sensible.

A more detailed description of the projects and the contents of their packages can be found in Appendix E.

4.6.2 Other Changes not mentioned yet

We have described the implementation of the main features that were added to the IDE, but certain other noteworthy modifications to Palkus' code have not been mentioned yet. These are:

1. The Curry grammar used in the IDE was modified in multiple places.

   ▷ Previously, the parser generated from the grammar was only used to parse complete modules. As part of the type-checking functionality, a parser for type-expressions was required. It was possible to use the existing parser for the respective rule in the grammar for this purpose, with one small change: The annotation defining the so called *hidden tokens* was removed from the *Module* rule and added globally instead. *Hidden tokens* are tokens like whitespace that the parser will ignore. Without this change, the parser for type-expressions would not ignore the hidden tokens and consequently would fail in many cases where it should not.

   ▷ In some places the grammar was correct, but the syntax tree generated did not contain all the information contained in the input. For example, one alternative in the rule for patterns was:

   ```
   {LiteralPattern} Literal
   ```

   The way this is notated, the syntax-tree generated for the literal would be just a leaf node *LiteralPattern*, without any parameters. But for the type-checking implementation, we need access to the actual literal to determine its type. Hence, this alternative was changed to:

```
{LiteralPattern} literal=Literal
```

Now, the node *LiteralPattern* would have a member called `literal` which contains the actual literal. Changes such as this one were necessary in several places.

▷ The grammar contained ambiguities in three cases. Notably, in one case involving type-expressions the ambiguity was present in the Curry report [Mic12]. The corresponding grammar rules in the report were:

$$\begin{aligned}
\mathit{TypeExpr} &::= \mathit{SimpleTypeExpr}\ [\rightarrow \mathit{TypeExpr}] \\
\mathit{SimpleTypeExpr} &::= \mathit{QTypeConstrID}\ \mathit{SimpleTypeExpr}_1\ \ldots\ \mathit{SimpleTypeExpr}_n \\
&\quad \mid \mathit{TypeVarId} \\
&\quad \mid \texttt{()} \\
&\quad \mid \texttt{(}\ \mathit{TypeExpr}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{TypeExpr}_n\ \texttt{)} \\
&\quad \mid \texttt{[}\ \mathit{TypeExpr}\ \texttt{]} \\
&\quad \mid \texttt{(}\ \mathit{TypeExpr}\ \texttt{)}
\end{aligned}$$

The ambiguity arises from the recursion in the first alternative for `SimpleTypeExpr`. Consider the input `A B C`. This is a valid type-expression with two

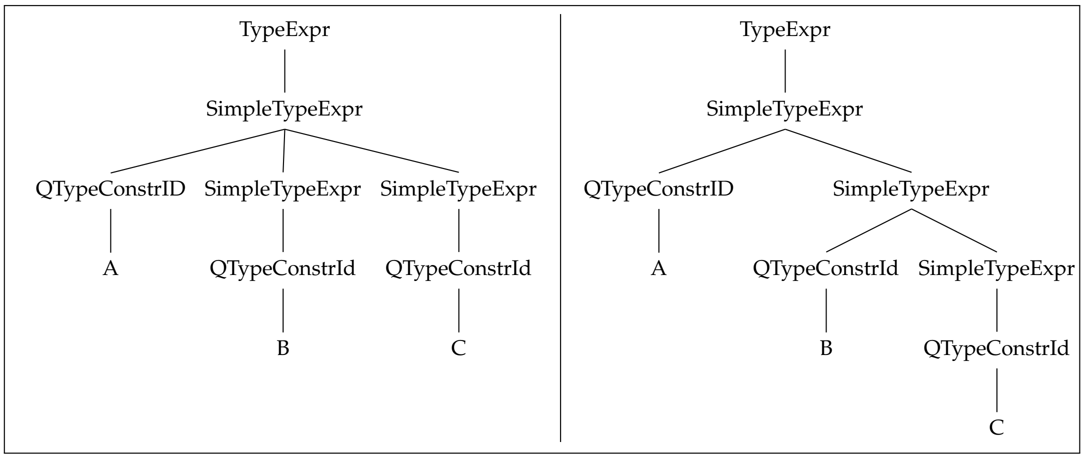


Figure 4.9. Productions for the type expression `A B C` in the unmodified grammar

productions, which are shown in Figure 4.9. This ambiguity is removed by adding an additional rule in the following manner:

$$\begin{aligned} TypeExpr &::= TyConExpr\ [\rightarrow TypeExpr] \\ TyConExpr &::= QTypeConstrID\ SimpleTypeExpr_1\ \ldots\ SimpleTypeExpr_n \\ &\mid SimpleTypeExpr \\ SimpleTypeExpr &::= TypeVarId \\ &\mid \texttt{()} \\ &\mid \texttt{(}\ TypeExpr_1\texttt{,}\ \ldots\texttt{,}\ TypeExpr_n\ \texttt{)} \\ &\mid \texttt{[}\ TypeExpr\ \texttt{]} \\ &\mid \texttt{(}\ TypeExpr\ \texttt{)} \end{aligned}$$

It should be noted that the grammar, as implemented, still lacks certain features described in the Curry report. Additionally, certain rule names deviate (somewhat unnecessarily) from the names of the rules in the report, which makes it harder to compare the two.

2. Scala was integrated into the projects; many of the new features are implemented in Scala, and even small parts of the existing code where refactored to scala in order to clean the code up and to remove bugs. The reason for adding Scala is its expressiveness. Scala allows for a functional programming style (using pattern matching, for-expressions and anonymous functions) which allowed for shorter and more easily maintained code. This was especially relevant for implementing the type inference algorithm.

   However, adding Scala did not come without its caveats and problems. These are described in detail in Section 5.1.

3. The *Curry Project Explorer* implemented by Palkus is supposed to replace the default project explorer provided by Eclipse. It was removed because it provides (at least in its current state) no benefit to the usability of the project; rather, it even lacks certain features. For example, the context menu does not provide certain items such as "delete" and "rename", and non-Curry files are not displayed correctly in the directory tree. In general our suggestion is to use and extend the existing explorer instead of creating a new one. For now, the *Curry Project*

Explorer is removed from the GUI by deleting the extension point connection, but the respective source code was left in place.

Chapter 5

Experiences

In this chapter we will describe our experience with the tools used in the development of the Curry IDE, and give some hints that hopefully save some time for future developers or maintainers.

Generally, the things that work well are not noticed; only when something breaks it calls for our attention. Hence, this chapter may involuntarily sound rather negative, because it will focus on the things that did not work as expected. This is not the intended message; all in all we are happy with the tools used.

5.1 Scala

Our initial reasoning for adding Scala to the plugin was that it allowed for a much shorter and more concise implementation of the type inference algorithm, which is functional in nature. Looking at the actual implementation, it is apparent that the functional features such as first class functions, abstract data types and pattern matching were indeed used frequently. And Scala was also used in several other places including the code implementing the Curry console and parts of the debugging functionality.

Unfortunately, there was a surprisingly large amount of problems connected to the use of Scala, even though the two languages interact just fine in theory.

5.1.1 Interaction between Scala and Xtext

Xtext is a Java and not a Scala project. Xtext (or its developers) expect all source files to be either Java or Xtend (an extension of Java). When executing the MWE2 workflow to automatically generate code, the existing source files are analyzed. Our guess is that this is necessary for example to configure the dependency injection with google guice.

When adding Scala source files to a project, it is possible to reach a state, where, even though there are no actual (compile-time) errors in the code, it is not possible to actually build the project. The exact conditions for this scenario are:

▷ The project is completely “cleaned”, i.e. all compiled classes (for both Java and Scala source files) are removed, as well as all the classes being automatically generated as part of the MWE2 workflow.

▷ There is at least one Scala class that depends on any Java class that is to be generated automatically by the MWE2 workflow.

  An example for such a dependency is when we use any of the classes representing the Curry syntax tree from a Scala class.

Under these circumstances, the following cycle appears:

▷ The Scala compiler fails when encountering the missing dependency.

▷ The MWE2 workflow fails, because it can not handle the compilation error in the Scala classes, and it can not parse the Scala class directly.

Note that the MWE2 workflow can handle compilation errors just fine, as long as they happen in Java classes. Our guess is, that in these cases, the Java sources are simply parsed, ignoring the semantic problems.
This problem was especially annoying because it will not be noticed unless the exact preconditions are met - it even might have gone unnoticed until somebody else checked out the project into a clean directory and tried to build it. Particularly:

▷ If there exist compilations (regardless of whether these are up-to-date) of the relevant Scala sources, the MWE2 workflow will succeed (presumably by retrieving all necessary information from said compilations).

▷ If the auto-generated sources do exist (even if not up-to-date), the Scala classes may - under the right circumstances - compile without a problem.

Because this problem is inherent to the process of Xtext and MWE2, we decided not to try to fix it or to apply any tricks (like committing the automatically generated sources to our repository). Making the steps for building the system even more complex would make it (even) harder for new maintainers to start working on the system. Instead, we simply avoid the problem by not using Scala for any classes that need a dependency on any automatically generated classes. In practice this mostly means that Scala could not be used to examine the syntax tree for Curry modules, those parts of the implementation were written in Java.

5.1.2 Interaction between Scala and Java

In most cases, the interaction between Scala and Java worked just fine; for example accessing Java classes from Scala and vice versa caused no problems in general. Yet, there are some problematic cases.

- ▻ The first problem is more of a nuisance, but still should be mentioned as it might confuse a programmer new to the system: If the source code is modified and contains errors, it is possible that the Scala compiler (or more precisely: the Scala build tool, sbt) will crash, possibly causing a cascade of other errors. Fixing the initial error will remove the problem, so the errors caused by the crash can be ignored. Sometimes, the initial error might be hard to spot in the flood of error messages, though.
- ▻ In one case, there was a weird problem when accessing a specific Java class (or its methods) from Scala code. The methods are

  ```
  PlatformUI.getworkbench()
  PlatformUI.getActiveWorkbenchWindow()
  ```

  Our workaround was to create a facade class [1] in Java that basically adds one layer of indirection, which removed the problem.
- ▻ The type cache is saved persistently in order to retain the typing information when Eclipse is closed and re-opened.
  The type cache data structure is implemented in Scala, using Scala collection classes such as `scala.collection.mutable.Map` as well as custom made classes for the representation of types.

  For some reason, deserialization failed when directly serializing this data structure. We tested the problem and observed:

  - ▻ Java collection classes containing custom Scala classes, e.g.

    ```
    java.util.LinkedList<MyScalaClass>
    ```

[1] `uinonxtext.curryconsole.ScalaWorkaroundHelper`

would deserialize without a problem.

▷ If no single custom Scala class was involved in the complete data structure (but any number of classes from the Scala libraries), e.g.

```
scala.collection.immutable.HashMap[
  Int,
  scala.collection.immutable.List[MyJavaClass]]
```

then deserialization succeeded.

▷ otherwise, i.e. if a custom Scala class was used inside any Scala collection class, the deserialization would fail, with some "class not found" error.

The solution for this problem is to transform any custom class into a tuple (which is a class from the Scala library, and therefore causes no problem) before serialization; and reverse the transformation after deserialization.

▷ Accessing Java classes from Scala was always possible without any problems. Sometimes, it is necessary to transform the collections into their Scala-equivalents, but this is a simple step. In most cases, the reverse, i.e. accessing Scala code from Java code, was unproblematic. Two things should be mentioned though.

Firstly, to access Scala *objects*, a special syntax must be used. In Scala, classes can have no static members; instead, classes can have a "companion object" which is a singleton, i.e. it can not be instantiated and it has one static instance. Let us assume we have one such object `foo.Bar`. To access this object from Java, we need to write (see, for example, [web:objjs]):

```
import foo.Bar$;
[..]
foo.Bar$.MODULE$
[..]
```

Apart from looking a bit unusual, this works just fine.

Secondly, using Scala inner classes as part of a Java interface was not possible. This problem materialized when defining the interface for expression debuggers. The Scala class `DebugPoints` contains an inner class `DebugPoint` representing the debugpoints the user can add and use for debugging purposes in the IDE. One method in the interface for expression debuggers is supposed to be a `DebugPoint`, but it was not possible to refer to this type. To solve this, we chose to use the generic super-type `Object` from java as type of this parameter. This necessitates type casting when using the parameter, but seemed to be the simplest solution.

5.1.3 Interaction between Scala and Eclipse

Apart from the single problem accessing the `PlatformUI` class from Scala, everything worked fine when using Scala. When implementing extension points for the plugin, it is no problem if the implementing class for some interface is written in Scala. We should mention that, when editing the extensions definition for the plugin using the corresponding user interface (in contrast to editing the file `plugin.xml` directly), when using the dialog to define the class to use, Scala classes will not be listed. In this case, the programmer can simply insert the qualified name of the class by hand.

5.1.4 General Observations about Scala

A future developer might notice that even on current hardware, building the plugin takes a good amount of time (a complete build takes half a minute, roughly, on this machine). More importantly, small changes in the source code often require a rebuild that takes more than a few seconds. We will not provide objective measurements here, but our hypothesis is that scala caused a large increase in build times. In an older version of the IDE's source code that did not contain Scala yet, the builds are almost instantaneous. The difference is large even when we consider that the older version has less functionality.

One difference between Scala and Java has not been mentioned: Scala has no *checked exceptions*. In Java, it is checked at compile time that checked exceptions are either handled (by `try .. catch`) or forwarded to the caller (which must be annotated with the `throws` declaration for the respective function). While Scala has exceptions and it is possible to `try .. catch`, all exceptions are *unchecked*: Functions do not have a `throws` declaration, and there is no compile-time error if any type of exception is not caught. While this design decision is understandable, there is a large risk that the programmer forgets to catch exceptions, as we noticed when refactoring certain parts of the IDE's code from Java to Scala.

5.2 Eclipse and Xtext

For the most part, Eclipse's plugin interface is well designed and well documented. It is possible to execute (and debug) the plugin from the IDE, which starts a second Eclipse instance. Eclipse supports run-time code replacement, so that changes become effective without restarting the second Eclipse instance running the CurryIDE plugin. Provided that the the user has enough memory (We encountered combined memory usage of up to 3GB), this allows for fast development iterations.

There exist, however, minor flaws which again can surprise the plugin's developer, and thus should be mentioned.

▷ One rather annoying problem appears when debugging the plugin. When launching the plugin as an Eclipse application (which opens a second Eclipse application), the process aborts randomly with an error.

```
"Launch Error: The application could not start.
               Would you like to view the log?"
```

When this happens, the launch can simply be repeated; this second launch was never observed to fail. The probability of this happening on the first try (after the last successful launch) is roughly one third.

This problem might be connected to Xtext, and not Eclipse. Because it only appeared for debugging (and not when the plugin is properly installed), we decided to ignore it (as much as possible).

▷ In certain cases, it is necessary to refer to extensions[2] from the source code. One trivial example is when the provider of the extension wants to access the corresponding implementations at run-time. In such cases, the extension's identifier (a string) must be used.

For example, to access the implementations we would use:

```
IConfigurationElement[] impls =
  Platform
    .getExtensionRegistry()
    .getConfigurationElementsFor(id);
```

[2]see chapter 2.2.3

where *id* might, for example, be

```
"de.kiel.uni.informatik.ps.CurryIde.expressiondebuggers"
```

Now, `impl` would contain a list of expression debugger implementations.

The programmer has to be rather careful with the identifiers of extensions, for two reasons:

▷ When the identifier is wrong (i.e. a non-existing identifier, not just a different one than expected, for example by a typo in the string), any access may fail silently.

For example, when there is a typo in the identifier, and we execute:

```
IConfigurationElement[] impls =
  Platform
    .getExtensionRegistry()
    .getConfigurationElementsFor(         // note the typo vv
      "de.kiel.uni.informatik.ps.CurryIde.expressiondebuggesr"
    );
```

no exception is thrown, and `impls` is *not* null, but an empty array.

When handling custom *markers*, using a wrong identifier will even still create a marker, just not of the correct type. This sort of issue is annoying to debug.

▷ Identifiers must be *fully qualified*, for some definition of *fully qualified*. For example, when accessing the marker extension that has, in the plugin definition (i.e. in the file `plugin.xml`), the identifier `curry.debug`, in the code, the string

```
"de.kiel.uni.informatik.ps.curry.CurryIDE.ui.curry.debug"
```

must be used. So the project's qualified name must be prepended. It might be a good idea to follow the advice in [ML05, p. 43] and to include the project's identifier in the extension's identifier.

▷ In addition to the inherent unsafety of the identifiers for plugins, we observed another unexpected behavior which we could not find documentation for. We tried to incorporate our new views into the *Show View* menu in the *Window* tab of the Eclipse IDE, so the views can be more easily opened. Even though this had already been working for the *Curry Project Explorer* and the *Curry Analysis View*, adding new views did not work. This seemed to depend on the identifiers we used for the views; for example when we changed the id of the *Curry Analysis View* (both in the definition *and* in the respective class), it was not available in the *Show View* menu, but reverting the change resolved the problem. The issue might be some kind of caching of the perspective in the workspace settings.

▷ To highlight debugpoints in the source code, our first idea was to use the marker specialization used for Breakpoints[3]. As it turns out, this is a bad idea, because the plugin that handles breakpoints in Eclipse expects all markers that are a sub-type of `breakpointMarker` to be connected to an implementation of `IBreakpoint`. Because we did not connect the markers to `IBreakpoint`, but to debugpoints, exceptions would occur. Additionally, because we do not actively use the Breakpoint functionality in the Curry IDE, these crashes would be seemingly random.

The underlying problem is that the documentation for Eclipse does not properly point out which markers can be used as super-types, and under which conditions.

▷ In context of markers, one might notice that markers are defined in the `ui` project, not in `uinonxtext`, which is contrary to our guidelines, as the DebugPoints are not related to Xtext and thus should be placed in `uinonxtext`. The reason for this is that the marker simply will not work if we define it in `uinonxtext`. Our guess is, that, given the way the extra project was added to the system, the new project *uinonxtext* is not visible to the code creating the marker. If we could explain the problem in more detail, we probably would be able to resolve the issue. We chose the simpler alternative and moved the marker definition to the `ui` project.

▷ Eclipse provides examples/"wizards" for different extensions points. Unfortunately, for many extension points, there are no examples; in other cases, the examples use extension points which are actually *deprecated*. Even the "hello world" example, which adds a button the UI which shows a dialog, uses the `ActionSet` extension point which is deprecated.

[3] `org.eclipse.debug.core.breakpointMarker`

5.3 Guice Dependency Injection Framework

Google Guice is very useful and easy to learn. There are a few items to note, though. The first two relate to the interaction between Scala and Guice:

- ▻ When an new instance of a class containing `@Inject` annotations is created, the Guice *injector* assigns some instances to the corresponding members. This means that in Scala, all injected members must be assignable, i.e. they must be declared as `var`, even though they should be treated as immutable after the injection.
- ▻ Scala's objects are (non-lazy) singletons, i.e. they have exactly one static instance. Because the injection of dependencies normally happens at construction, we can not use the normal procedure when a Scala object contains `@Inject`-members. In such cases, it is possible to manually inject the Scala objects by creating and applying an injector in the `start` method of the `CurryActivator` class. [4]

In chapter 2.3.3 we mentioned the `@Singleton` annotation. For any class with this annotation, each injector creates only one instance of that class and uses it for all injections[5]. An important consequence is that with multiple injectors, there *can* be multiple instances of classes with this annotation, which might be surprising considering the name "singleton".

Extension-points can name classes that contain the corresponding implementation. In such cases, the classes will be instantiated when the extension is needed. Without any modifications, this instantiation would not execute injections for the new instance. If the latter is desired, it is necessary to modify the entry referencing the class, and prepend the project's *ExecutableExtensionFactory*.

Instead of

```
de.kiel.uni.informatik.ps.curry.uinonxtext.debug.impl.TraceValueDebugger
```

we would write (as one long line)

```
de.kiel.uni.informatik.ps.curry.uinonxtext.ExecutableExtensionFactory:
  de.kiel.uni.informatik.ps.curry.uinonxtext.debug.impl.TraceValueDebugger
```

[4] An example of this can be seen in the class `de.kiel.uni.informatik.ps.lcurry.ui.LCurryActivator`

[5] see `https://google-guice.googlecode.com/git/javadoc/com/google/inject/Singleton.html`

This is easy to forget; the consequence will generally be a null-pointer-exception when the non-injected member is accessed.

5.4 Miscellaneous Hints for Future Developers

▷ To manage our source code, the version control system *git* was used. When working on the project, it is common to switch between different versions of the source code (using git's functionality, e.g. by switching to different branches or commits, or by using the stash). In such cases it is necessary and important to properly refresh Eclipse and rebuild the project.

The developer should be aware that the complete process consists of three steps:

1. The relevant projects must be *refreshed* in Eclipse, to make Eclipse aware that the file system has changed and must be re-read. The easiest way to do this is to select the projects in the *package explorer* and press `F5`.
2. The MWE2 workflow must be executed in order to refresh the automatically generated sources in the project. In some cases, this step can be omitted.
3. The project must be re-built. Depending on the IDE's settings, this step might be executed automatically.

▷ It is possible to reach states where the Curry IDE either does not build or throws exceptions at launch. In many cases, full cleans help.

In one instance, when we tried to refactor the `ui` project, an even worse situation was encountered. The project would build without errors, but when launching the Curry IDE in debug mode, innumerable exceptions are thrown. The corresponding stacktraces showed no direct involvement of the Curry IDE's code.

In that case, cleaning the project did not work. Switching to any other revision using *git* did not help. Manually cleaning the project directory (by deleting the directory and cloning the git repository again) did not help. Deleting the workspace used by the Curry IDE did not help. Instead, it was necessary to delete certain configuration items in the *settings* directory of the workspace *that holds the plugin's projects*. The exact steps to remove this problem are, assuming that `workspace` is the directory of the workspace containing the plugin's projects:

1. Close Eclipse;

2. Delete the directory
   `workspace/.metadata/.plugins/org.eclipse.core.resources;`
   This step removes the problem; unfortunately it also removes all projects from the workspace. Hence;
3. Re-import the projects of the Curry IDE (and any other that were present) to the workspace.

Needless to say that debugging such a problem is annoying and time-consuming. If the developer ever encounters any strange problems, especially after reverting to a state that previously worked, he/she is strongly advised to create a second, fresh workspace and import the projects into it in order to test if the workspace settings are responsible.

▷ In most cases, imports can be automatically added adding new code to a class. When accessing Scala objects though (using the `foo$.MODULE$` notation), this breaks. In such cases it is necessary to manually fix the list of imports; if necessary, both `foo` and `foo$` must be imported.

▷ In order to compare Palkus' IDE with the current IDE, it is highly advised to clone the git repository twice, and use two Eclipse workspaces, in order to avoid all the problems arising from switching revisions.

▷ One topic in plugin-development is important but easily over-looked: Threads in the Eclipse IDE. Often, large parts of the code are executed in the UI-thread, so problems regarding thread-safety will not be noticed. But in general, Eclipse does of course use multiple threads, and the code in plugins might run in different threads. Additionally, certain operations are only allowed in the UI-thread in Eclipse, which is important if code can be executed in a non-UI context. While we kept this topic in mind, the Curry IDE might contain "silent" problems connected to thread-safety.

Chapter 6

Known Problems and Future Work

6.1 Functional Deficiencies and Bugs

6.1.1 The Curry Grammar

In Section 4.6.2 we discussed certain changes to grammar used for Curry in the IDE, and we mentioned that the names of the rules are inconsistent with the grammar in the Curry report. This should be fixed to allow, as the next step, to complete the grammar in the IDE. For example, *record syntax* is currently not supported; additionally, new language elements such as *type classes* were recently introduced and require certain changes to the grammar.

6.1.2 Syntax Error-Messages

The error messages created by the parser generated with ANTLR are rather hard to understand in many cases. When describing the feature in Section 3.2.4 we used the faulty code

```
answer = (6*(5+2)
```

which contains a superfluous parenthesis and produced a perfectly understandable error message "expected ')'" located at the end of the expression. But, if we removed one set of parenthesis, i.e.

```
answer = 6*(5+2
```

then the error is placed at the ∗-operator with the message "mismatched input `'*'` expecting RULE_END_OF_LINE". We are not certain if or how much these messages can be improved while using the automatically generated parser, though.

6.1.3 The Modified Token Stream

In order to implement the layouting in Curry, Marian Palkus modifies the token stream (i.e. the output from the lexer) and creates certain "virtual" tokens that are used for correctly implementing the grammar. To implement the debugging, we make a modified copy of the source-code, and (indirectly) print all the tokens of the original source code. This revealed a bug: Certain virtual tokens are not printed as empty strings; instead, a single character of the last token is repeated. We tried but failed to find the cause of this problem; instead, we simply check for these tokens and do not print them.

While this workaround seems to work (the printed code for debugging did not contain syntax errors anymore), it can be observed in certain cases. For example, when retrieving the type of expressions, the output will sometimes contain additional characters. To properly address this problem, the underlying bug must be fixed.

6.1.4 Typechecking

Syntactic Features

The type-checking code is not completed for all syntactic features in Curry. For example, certain notations for lists are not supported yet. In these cases, no type equations are produced for the respective type-equations and consequently, type errors might be missed. Also, the IDE will not be able to provide a result when the user requests the type of a specific sub-expression.

Operator Precedence

There is another more important problem with type-checking: Operator precedence is not properly implemented. The syntax tree generated in the IDE does not respect operator precedence, because these precedences are dynamic in Curry. In the Curry compilers, expressions are re-structured after parsing to create syntax trees with correct precedence, but this step is not implemented in the Curry IDE. For syntax-checking, this creates no problem, because any source is syntactically valid exactly if the parser that ignores precedences succeeds.

However, for type-checking, we need the correct syntax tree, because we need to apply the types of the operators in the correct order.

The expression `1+2 == True` is type correct, with the implicit structure `(1+2) == True`. On the other hand, if we use wrong precedences, i.e. read it as `1 + (2==True)`, then the expression is not type-correct.

The only way to fix this problem is to do the same as the Curry compilers, i.e. to re-structure expressions after parsing according to the fixity-declarations.

Multiple Equations

If a function consists of multiple equations, the different equations are currently not connected when doing type-checking. This does not matter if there is a type signature, but if there is non, the different equations may each be consistent but still not compatible. The latter case is not detected currently.

```
foo x = (x + 1) -- foo :: Int -> Int, (locally correct)
foo x = x ++ "a" -- foo :: String -> String, also locally correct
```

but we cannot unify `Int->Int` and `String->String`, so the complete function is not type-correct. This is not detected in the IDE.

6.2 User Interface

6.2.1 Curry Analysis View

The user interface can be improved in several cases. One example is the *Curry Analysis View* that is very basic at the moment. It might be useful to allow the results to be sorted by the Curry function names, and marking the results as out-dated if the respective code has changed (which might be hard to determine because analysis might not be local).

6.2.2 The Explorer

We removed the *Curry Project Explorer* from the interface, because it was too buggy in its presentation. Still, certain elements, like the wizards for Curry modules, are not directly accessible in the default project explorer at the moment. While this does not prevent the creation of new modules (the user can either access the wizard via the menu bar, or simply create a `.curry` file), the usability of the project explorer can certainly be improved.

6.2.3 Interoperability with other languages

The features of the Curry IDE are meant for Curry, but the user might want to have an Eclipse installation that works for multiple languages. For example, we do not want a Curry-specific context menu item when editing Java source-code.

To implement this, it is necessary to take the type of the file being edited into account, as well as the opened perspective. When developing the Curry IDE, we did not focus on such things, so in the future it should be tested if any Curry-specific items in the interface are visible when they should not be.

6.3 Testing

While we are not proud of the fact, we feel obliged to admit that there is a distinctive lack of testing of both the features we added to the IDE and of those our predecessor introduced. However, it should be mentioned that many features that involve the user-interface, are hard to test, because simple unit tests are not feasible.

6.4 Future Work

6.4.1 Better Usability for CASS

To improve the usability of the analysis integration, it should be possible to start the analysis server from within the IDE. For this purpose, an *external tools* configuration could be used. This addition could then also remove the necessity to restart Eclipse to refresh the connection to the analysis server, as the connection could be automatically refreshed when launching the external tool.

6.4.2 Integration of the Debugging Module in the IDE's Feature

Currently, the Curry library `EclipseDebugWriter` is not integrated directly in the feature. Consequently, it must be distributed separately; and additionally, its path must be added manually to a project before the debugging functionality will work. It should be possible to automate these steps.

6.4.3 Ideas for New Debuggers

We have two ideas for new kinds of debuggers. The first idea is to improve the existing debugger for observations. Currently, it is necessary to start an external user-interface to display the results of this kind debugger. It would be nice if this functionality was integrated directly into the IDE. This would require that the IDE interprets and displays the data written by the COOSy library when debugging. As we mentioned above, we started implementing this feature by slightly modifying the COOSy library in order to use the debugging TCP stream as output, but we can not display the results in the IDE yet.

A second idea is a truly new type of debugpoint, inspired by the imperative *breakpoint*: It is possible to implement breakpoints in Curry, i.e. to interrupt the execution of the program temporarily in order to inspect its state. While it would not be possible to freely inspect any variables, the user could combine such a functionality with observations in order to gather additional information about the order of the evaluation of specific expressions. To implement this idea, the debugging functions would not only *send* messages over TCP to Eclipse, but also wait for a *reply*. The user can control when the reply will be sent, and thus controls when the execution of the Curry program continues. Unfortunately, because the TCP stream would be used in a different fashion than for the other types of debuggers, this addition would require a slight change to the debugger interface.

Chapter 7

Conclusion

The Curry IDE plugin for the Eclipse IDE, originally programmed by Marian Palkus, was extended by two important features: Type-checking and observational debugging. For the latter, we introduced a extension point that allows future additions. Furthermore, the IDE was improved in many aspects; we completed the integration of the external program analysis tool *CASS* and allow the user to launch the Curry compiler from within the IDE, parsing any error messages that might occur.

Our work took more time than initially expected. First and foremost, we encountered various unexpectable and time-consuming issues when integrating Scala; in hindsight – and in our opinion – the integration of Scala unfortunately was not worth the effort. Secondly, we needed to get familiar with many different topics: The existing code base, the Eclipse framework, the Xtext framework and details about the Curry language, to name the major ones.

On the other hand, while there is a significant learning period, once we were familiar with the Eclipse and Xtext frameworks, it was surprisingly easy to add new functionality.

We achieved important steps towards better usability of the plugin by fixing bugs in the interface and by bringing the user experience closer to the standard, as defined by other plugins for major languages like Java, C++ or Scala. Examples for the latter are the removal of the separate project explorer and the use of *run/debug* modes for launching (instead of the *external tool*).

On the other hand, we also tried to improve the quality of the plugin's source code by restructuring the project and by refactoring or rewriting certain classes. We hope that this, together with our description of the general working of plugins and the specific structure of the project, will allow future maintainers to get started quickly.

Bibliography

Literature

[Bra+04] B. Braßel et al. “Observing functional logic computations”. In: *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL’04)*. Springer LNCS 3057, 2004, pp. 193–208.

[CS09] Diego Cheda and Josep Silva. “State of the practice in algorithmic debugging”. In: *Electronic Notes in Theoretical Computer Science* 246 (2009). Proceedings of the 17th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2008), pp. 55–70. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2009.07.015. URL: http://www.sciencedirect.com/science/article/pii/S1571066109002370.

[Han13] Michael Hanus. Notizen zur Vorlesung: Deklarative Programmiersprachen. 2012/2013.

[HS14] M. Hanus and F. Skrlac. “A modular and generic analysis server system for functional logic programs”. In: *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM’14)*. ACM Press, 2014, pp. 181–188.

[Mic12] Hanus Michael. Curry: An Integrated Functional Logic Language (Vers. 0.8.3). Available at http://www.curry-language.org. 2012.

[ML05] Jeff McAffer and Jean-Michel Lemieux. Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications. Upper Saddle River, NJ: Addison-Wesley, 2005. ISBN: 978-0-321-33461-9.

[MM82] Alberto Martelli and Ugo Montanari. “An efficient unification algorithm”. In: *TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS (TOPLAS)* 4.2 (1982), pp. 258–282.

[OSV08] M. Odersky, L. Spoon, and B. Venners. Programming in Scala. Artima, 2008. ISBN: 9780981531601. URL: http://books.google.de/books?id=MFjNhTjeQKkC.

[Pal12] Marian Palkus. An Eclipse-Based Integrated Development Environment for Curry. Diploma Thesis. 2012.

[Pop06] Bernard James Pope. A Declarative Debugger for Haskell. Doctoral Thesis. 2006.

Online Resources

[web:antlr] Terence Parr. ANother Tool for Language Recognition. URL: http://www.antlr.org/.

[web:cass] Michael Hanus. CASS: A Curry Analysis Server System. URL: http://www-ps.informatik.uni-kiel.de/currywiki/tools/cass (visited on Mar. 20, 2014).

[web:coosy] Micheal Hanus. The Curry Object Observation System. URL: http://www.informatik.uni-kiel.de/~pakcs/COOSy/ (visited on Mar. 20, 2014).

[web:curryimpls] Michael Hanus. Implementations of Curry. URL: http://www-ps.informatik.uni-kiel.de/currywiki/implementations/overview (visited on Mar. 20, 2014).

[web:eclwiki] FAQ Where did Eclipse come from? URL: http://wiki.eclipse.org/index.php?title=FAQ_Where_did_Eclipse_come_from? (visited on Feb. 27, 2014).

[web:ggsngltn] javadoc for com.google.inject.Singleton. URL: https://google-guice.googlecode.com/git/javadoc/com/google/inject/Singleton.html (visited on Mar. 23, 2014).

[web:gguice] google-guice. URL: http://code.google.com/p/google-guice/ (visited on Mar. 23, 2014).

[web:hood] The Functional Programming Group. HOOD. URL: http://www.ittc.ku.edu/csdl/fpg/software/hood.html.

[web:kics2] Michael Hanus. KiCS2: Compiling Curry to Haskell. URL: www-ps.informatik.uni-kiel.de/kics2/ (visited on Mar. 23, 2014).

[web:mcc]	Wolfgang Lux. The Münster Curry Compiler. URL: `http://danae.uni-muenster.de/~lux/curry/` (visited on Mar. 23, 2014).
[web:objjs]	@marius @stevej and @lahosken. Java + Scala interoperability. URL: `http://twitter.github.io/scala_school/java.html` (visited on Mar. 11, 2014).
[web:osgi]	OSGi Alliance. URL: `http://osgi.org` (visited on Mar. 23, 2014).
[web:pakcs]	Michael Hanus. PAKCS: The Portland Aachen Kiel Curry System. URL: `http://www.informatik.uni-kiel.de/~pakcs/` (visited on Mar. 23, 2014).
[web:scala]	URL: `http://www.scala-lang.org/` (visited on Mar. 11, 2014).
[web:w3xml]	World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Fifth Edition). URL: `http://www.w3.org/TR/2008/REC-xml-20081126/` (visited on Mar. 23, 2014).
[web:xtend]	URL: `http://www.eclipse.org/xtend/` (visited on Mar. 12, 2014).
[web:xtext]	URL: `http://www.eclipse.org/Xtext/` (visited on Mar. 12, 2014).
[web:xtextrefs]	itemis Leipzig. Xtext Cross References and Scoping – an Overview. URL: `http://blogs.itemis.de/leipzig/archives/776` (visited on Mar. 23, 2014).

Appendix A

Glossary/Definitions

REPL *Read-Eval-Print-Loop* - refers to an interactive command-line session of an interpreter/compiler where the user can evaluate expressions or execute statements of the respective language. The result is printed, and the the user can input a new expression. In most cases, it is possible to load and use libraries in a REPL. On the other hand, not all languages support the declaration of objects (e.g. functions) that can be used in subsequent commands: The Scala compiler does support that, but the interpreters for Haskell and Curry (i.e. Ghci, PAKCS and KICS2) do not. Consequently, the syntax allowed in the REPL is only a subset of the respective language, in those cases.

project Might informally refer to the whole Curry IDE, but has a specific formal meaning when talking about *projects in Eclipse*, see Section 2.2.1.

plugin Just like the term *project*, the Curry IDE may informally be called a *plugin*, but the term has specific meaning in Eclipse terminology, see Section 2.2.2: Eclipse is a *feature* consisting of a number of *plugins*.

feature In Eclipse terminology, the additional "packages" that provide new functionality, such as the Curry IDE or the Scala IDE, are called *features*. See Section 2.2.2

MWE2 *Modeling Workflow Engine 2* - one of the tools used by Xtext; used to manage the automatic generation of source code, for example the parser generator. See Section 2.3.2

EMF Eclipse Modeling Framework - used by Xtext to control the automatic generation of Java sources.

IDE *Integrated Development Environment*

EBNF *Extended Backus-Naur Form* - meta-language for the notation of context-free grammars.

UI *User Interface*

GUI *Graphical User Interface*

API *Application Programming Interface*

xml *Extensible Markup Language* - a markup language both human-readable and machine-readable. Used in various cases as the format for configurations in Eclipse.

build/builder In Eclipse, a *build* creates some form of product from the files in a project. In most cases, a build will execute the compiler of the respective language (and a linker, if necessary). For Curry, the builder updates the type cache, but does not invoke the compiler.

lazyness lazy evaluation is an evaluation strategy which delays the evaluation of expressions (i.e. it is non-strict) and avoids multiple evaluations of the same expressions by *sharing*. Used by Haskell, and in a modified fashion (see *needed narrowing*) by Curry.

needed narrowing The evaluation strategy used by Curry. This strategy, while distinct from Haskell's, can be called lazy because it is similarly non-strict and uses sharing. Additionally, it defines how free variables are handled.

ANTLR *ANother Tool for Language Recognition* - The *parser generator* used by Xtext and thus indirectly used when developing the Curry IDE.

perspective User-interface concept in Eclipse, used to describe a configuration of language- or topic-specific elements (*views*). See Section 2.2.4

view Sub-window inside the Eclipse IDE. See Section 2.2.4

Appendix B

Framework and Library Versions

B.1 Eclipse and its Plugins

Eclipse IDE	4.3	(Kepler)
Eclipse SDK	2.0.0	
Scala IDE	3.0.1	
Xtext SDK	2.4.2	
Xtend M2E extensions	2.4.2	
MWE 2 language SDK	2.4.0	

B.2 Curry Compilers and Tools

PAKCS	1.11.3	
KICS2	0.3.0	
SWI-Prolog	6.6.0	(used by PAKCS)
GHC	7.6.3	(used by KICS2)

B.3 Miscellaneous

Scala	2.10.2	
Java	1.7.0	
OpenJDK Runtime Environment	2.4.5	(64-Bit, IcedTea)

Appendix C

User Guide

C.1 Plugin Installation and Basic Setup

Preconditions: We expect that the user's system has working installations of the following software:

- ▻ A Curry compiler
- ▻ Java runtime
- ▻ The Eclipse IDE; he most basic package, i.e. "Eclipse Standard" from `https://www.eclipse.org/downloads/` should be sufficient.

We will further assume that the *update site* containing the Curry plugin feature is available either locally or online.
The installation of the plugin consists of the following steps:

1. Start the Eclipse IDE. The user will have to choose a workspace to use. Even though the workspace does not matter for installation (i.e., the installation will not be workspace-local) the user might want to create a separate workspace for Curry projects already.
2. In the *Help* menu, choose the option "Install new Software".
3. Add the update site of the Curry plugin. In the dialog, choose the option *local* and locate the update site folder (in the git repository, the folder is named `CurryIdeUpdateSiteBin`). Of course, future maintainers might choose a different location or use an archive.
4. Select the newly added update site. Select the plugin and press "next".
5. Confirm everything.

6. Restart Eclipse, for good measure. The plugin should now be installed.

7. A first step for configuring the Curry IDE is to define that Curry library settings. This can be done in the Eclipse preferences, in the *Curry* item. You should enter the `lib` directory in the user's PAKCS or KICS2 installation.

   The library can be assigned to Curry projects individually. The *default library* will be used as the initial value when creating new Curry projects.

C.2 Creating a Hello-World-Curry-Project

For this and the following descriptions we will assume that the Curry plugin is installed in the Eclipse IDE.

1. Create a new project. The respective wizard can be started via the menu *File/New/Curry Project*. After this step, there should be a new, empty project in the workspace.

2. Add a module to the project. Either select the project and use the menu *File/New/Curry Module* to create the module `Main` or right-click the project and create a file `Main.curry`

3. Open `Main.curry` and write

   ```
   main = putStrLn "Hello, World!"
   ```

   Note that syntax errors (if you make any) should be highlighted at this point.

   If identifiers from the standard libraries (including `Prelude`) can not be resolved, the project probably lacks appropriate Curry library settings. To fix this, first make sure that Curry libraries are defined globally (as described at the end of the last section); then, choose the Curry library from the project's preferences (in the *Curry* sub-item).

C.3 Launching a Project

To launch the sample program from the last section, you can either right-click on the project or on a specific module, and select *Run As/Curry Module* from the context menu. This will automatically generate a new run configuration with the default *run-time-command* set to "PAKCS".

If a specific modules is selected when launching, this module will be loaded automatically into the Curry compiler. Furthermore, this preference will be stored in the launch configuration, so that future launches will load that module, unless a different module is explicitly chosen. If no module was automatically loaded, the user can load the module `Main` by typing ":l Main" in the Curry REPL.

The REPL will be opened in the console view. For our example, assuming that module `Main` is loaded, we can execute the `main` function:

```
Main> main
Hello, World!
Main>
```

C.4 Typechecking and Analyses

CASS is used both to manually execute analyses and to retrieve typing information. Consequently, the user must start the analysis server before starting the Eclipse IDE, if he intends to use these features. Because typing-information is cached, type checking might work to a certain degree if the analysis server is not running, but in general (and especially for any functions that are newly added or modified), type-checking will stop working.

To execute analyses, the context menu in the editor can be used, i.e. the user must open a specific module and right-click in the editor view. "Curry Analyses" will be one item in the context menu. When a specific function is selected, the specified analysis will be executed for that function only; the output will contain only the result. On the other hand, if no specific function was selected, the analysis will encompass all elements of the current module. In that case, the output will list the functions in the module and the result for each one.

Note that the type-checking functionality uses a type-cache that is only updated when the project on project builds. Only the next type-check after the build will use the updated cache. Consequently, sometimes two extra steps are necessary to retrieve error messages:

Assume, we add, without intermediate builds, the following lines:

```
a :: Int
a = "abc"
```

While editing, the type cache will not yet contain any information about `a`; the type-error will not be recognized. To retrieve the error, we need to firstly save the module (which will update the cache), and secondly make any change to the module (which will trigger a new type-checking run).

C.5 Using the Debuggers

Precondition for any kind of debugging is that the program compiles, i.e. if the compiler returns errors when starting in *run*-mode, the *debug*-mode will not work either (and it might print weird errors).

To implement the debugging, the Curry IDE internally uses two libraries. The first is called `EclipseDebugWriter` (see Appendix F) and contains some Eclipse-specific utility functions for debugging. This library currently is not integrated into the plugin itself, and must be installed separately (it is, however, in the git repository, in the folder `curry_src/eclipsedebugging`). The second library is the `Observe` library from the *COOSy* tool. It can be found in the sub-directory `tools/coosy/` in both the PAKCS and the KICS2 installation directory. The directory containing both modules must be added as *external path*s; otherwise, the debugging-launch will not work.

To add the *external path*s, choose the corresponding tab in the Curry preferences *of the Curry project that is to be debugged*. Then, add the directory containing `EclipseDebugWriter.curry` and the directory containing `Observe.curry`.

The IDE provides different kinds of debuggers, but the basic process of creating the debugpoints and launching in debug-mode is the same:

1. Select an expression that you want to somehow trace or observe at run-time.

2. Right-click the selection, and, in the context menu, choose “add debugpoint for selection”. This will open the the *Curry Debugpoints* view.

3. Choose the type of debugger to use for the new debugpoint by using the drop-down menu in the list of debugpoints.
4. Launch the Curry program in debug-mode; this will work the same way as the run-mode, the only difference is to choose "Debug as" instead of "Run as". This will open a debugging console that works just as the normal console.
5. Execute any functions of your program. If any expression that has a corresponding debugpoint is evaluated, some form of side-effect will trigger.

C.5.1 Traces

When the type of expression debugger is set to *trace*, the side-effect is a simple notification in the *Curry Trace Debugging* view. On the other hand, if the type is *traceValue*, the value of the respective expression will be evaluated by converting it to a string (using `show`). This value will also be added to the list of events in the *Curry Trace Debugging* view.

Note that it is possible to highlight the expression connected to any of the events in the *Trace Debugging* view by double-clicking the line.

C.5.2 Observations

To view the observations recorded when using the *observe* type of debugger, the user has to open the corresponding program. This can be done by typing ":coosy" in the Curry REPL of the debugging session. In the tool, the user has to press the "refresh" button to get the actual output.

Appendix D

Plugin Development Guide

D.1 Setting up the Eclipse IDE

To set up Eclipse in order to start developing on the Curry IDE plugin, certain preconditions must be met. Firstly we assume that the same basic items as described in the user guide are installed, i.e. a Curry compiler, the Java run-time and the Eclipse IDE.

In addition to that, the development will require:

▻ The Scala binaries (from `http://scala-lang.org/download/`). We will assume that Scala is set up in a way so that the Scala compiler is on the system `PATH`.

▻ A series of plugins in the Eclipse IDE:

The *MWE 2 language SDK*

We used the update site at `http://download.eclipse.org/modeling/emf/emf/updates/releases/`

▻ *Scala IDE for Eclipse* and *Scala IDE for Eclipse dev support*

We used the update site at `http://download.scala-ide.org/sdk/e38/scala210/stable/site`

▻ *Xtext SDK* and *Xtend M2E extensions*

Update site at `http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/`

It should be noted that it might be preferable *not* to install the Curry IDE plugin in the installation of the Eclipse IDE that is used for development on the plugin, because this might cause conflicts when debugging.

With this setup, it should be possible to create both Scala applications and use Xtext. Next, we will see how the existing projects can be imported.

D.2 Opening the Curry IDE Projects

The first step is to choose a workspace to use for the development on the plugin; we recommend a separate workspace that will hold the various projects that constitute the plugin.

We assume that the developer has cloned the git repository, or has access to the source files (the folder containing the plugin's projects). These projects must now be *imported* into the workspace (in the menu `File/Import`). For development, we will need the five projects

▷ `de.kiel.uni.informatik.ps.curry.CurryIDE`

▷ `de.kiel.uni.informatik.ps.curry.CurryIDE.sdk`

▷ `de.kiel.uni.informatik.ps.curry.CurryIDE.tests`

▷ `de.kiel.uni.informatik.ps.curry.CurryIDE.ui`

▷ `de.kiel.uni.informatik.ps.curry.CurryIDE.uinonxtext`

With the initial setup, there will be compilations error, because the auto-generated sources are missing.

D.3 Building the Curry IDE

In general, a reasonable first step is to *refresh* all projects in the workspace, to ensure that changes in the file-system are reflected in Eclipse's caches.

The next step is to run the MWE2 workflow. This can be done by right-clicking on the file `GenerateCurry` in the package `de.kiel.uni.informatik.ps.curry` in the project `CurryIDE` and choosing *Run As/MWE2 workflow*.

When this step is finished, the project must be built. By default, a build will be executed automatically; otherwise, it can be started in the *project* menu of the IDE.

When the build is complete, there should be no more errors in the plugin's projects. There will, however, be a considerable amount of warnings.

Note that it is possible, especially when modifying Scala source code, that the build fails even though it should not. In such cases it often helped to execute a *clean* for all projects after the workflow and before the build. However, it is important not to start a *clean* while the workflow is executing, because the project is not properly locked and thus this will lead to errors.

D.4 Running the Curry IDE directly

The plugin can be tested directly: While developing, it is not necessary to deploy the plugin to test it, it can be launched from Eclipse, just like a custom application (for example, in Java) can be launched. It is possible to either use run-mode or debug-mode for these launches. To execute a launch, the user can right-click on any of the plugin's projects and choose *Eclipse Application* in the *Run As* or the *Debug As* menu.

The launch will start a second instance of the Eclipse IDE. In the launch configuration, the user can define the plugins that will be available in this IDE; depending on the setup used in the "parent" instance, the develop might want to disable certain features (because they are unnecessary in the Curry IDE and slow down the launch). Keep in mind, however, that the Curry IDE does depend on various other plugins that must not be removed.

When changing the source-code, a running instance of the plugin will be updated by Eclipse. In the cases where this is not possible (e.g. when the structure has changed too much), the user will be notified with a dialog.

D.5 Deploying the Curry IDE

Because we encountered several problems when deploying the project, we reverted to a rather "conservative" approach. This means certain steps might be unnecessary, but they certainly to not cause additional problems either.

1. Make sure that all plugin-projects compile successfully. To be absolutely certain, the user might consider to re-execute the workflow and clean and rebuild the projects.

2. In the CurryIdeUpdateSite, remove any contents but the `site.xml` file.

3. Open the `site.xml`. In the *Site Map* tab, remove any entries.

4. Then, create a new category with id and name "Curry". To that category, add the feature
   `de.kiel.uni.informatik.ps.curry.CurryIDE.sdk`.

5. Click "Build All"

6. Next, export the update site. To do this, right-click the CurryUpdateSite project, select export, and choose *Deployable features* in the *Plug-in Development* category. As the output folder, we use the `CurryUpdateSiteBin` folder in the git repository. In the options, it is imported to enable "Use class files compiled in the workspace". Now, execute the export.

7. The folder `CurryUpdateSiteBin` can now be used for installation in a separate Eclipse installation.

It should be noted that the `CurryUpdateSite` (*not* Bin) is accessible as an update-site itself. While installing the plugin is possible from this location as well, the plugin did not function reliably (because of unknown problems locating Scala classes at run-time).

Appendix E

Project Structure

E.1 project de.kiel.uni.informatik.ps.curry.CurryIDE

The core project that contains the specifications for Curry.

relative path	contents
`/`	The Xtext grammar file, the configurations for the MWE2 workflow, google guice dependency injection and and basic plugin configuration.
`debug.printer`	The printer that converts the syntax tree back source-code, used for debugging purposes. The `IExpressionCodePrinter` interface is used in the interaction between the printer and expression debuggers.
`description*`	Resource descriptions are an Eclipse concept for identifying the elements of a project. One example where this is relevant is when an import-statement is mapped to the actual file containing that module.
`formatting`	Theoretically used to define auto-formatting; practically empty at the moment.
`generator`	Theoretically used for automatic creation of resources (for example: an empty module template); empty at the moment.
`linking`	Manages the cross-referencing of items between multiple modules.
`naming`	Creates qualified names for the elements of modules.
`parser`	The modifications to the lexer necessary to handle Curry's layouting.
`scoping`	Scopes are used when resolving the cross-references of identifiers. For example, a locally defined variable has a limited scope.

relative path	contents
`typecheck`	The core of the type-checking functionality, plus relevant data structures.
`types`	The representation of the types in Curry, plus a parser to convert strings to these types.
`utils`	Miscellaneous helper functions
`validation`	The manual validations (i.e.: in addition to the default Xtext syntax-checking etc.) for Curry.
`lcurry*`	a different parser for lcurry (literate curry) files.

E.2 project de.kiel.uni.informatik.ps.curry.CurryIDE.ui

Contains all Xtext-specific UI-elements. Depends on the CurryIDE project. Compared to Palkus' IDE, this project is nearly unmodified; hence, we refer to his description of the contents: Appendix C.1.2 in [Pal12]

E.3 project de.kiel.uni.informatik.ps.curry.CurryIDE.uinonxtext

Contains UI-elements and front-end-functionality that is not connected to Xtext. Depends on both the CurryIDE and the CurryIDE.ui projects, but only uses one single helper class of the latter.

relative path	contents
`/`	Configuration of the plugin and for dependency injection; furthermore, the Curry perspective is defined here, and the `BuildCheck` class that handles the updating of the type-cache whenever a Curry project is built.
`analysis`	Several UI-elements for the analysis functionality and the implementation of the back-end, i.e. the communication with the analysis server.
`analysis.iface`	The interface for the visualizations for analyses.
`analysis.impl`	The class `CurryAnalysisTextVisualization` implements the interface above and features a simple text-based output for analyses.

relative path	contents
`curryconsole`	The console class (and certain relevant utilities) that is used when launching a Curry project (both for running and debugging).
`debug`	Similar to the setup for analyses, this package contains the UI-elements for the debugging functionality and the core implementation for handling debugpoints and for organizing the launch in debug-mode. Also contains the TCP server which receives and forwards the debugging output.
`debug.iface`	The interface for expression-debuggers consisting of two classes.
`debug.impl`	Six implementations of the expression-debugger interface; one of which (ObservationDebuggerExternal) is unfinished and not active, and one only an example (ZeroDebugger).
`explorerview`	The Curry explorer view which we deactivated. This view is not accessible by the Curry IDE's user anymore. We retained the respective classes to review its functionality.
`launch`	The Curry launch configuration, including the respective UI elements (the configuration's forms).
`preferences`	Both the global and the project-specific configuration for Curry.
`typeinferrer`	The utility that allows the user to retrieve the type of any expression in the Curry code. Does not contain the type-inference implementation.
`typing`	Higher-level management of the type-cache functionality; handles both the persistence of the type-cache and the mapping from projects to their specific caches.
`utils`	Some helper classes that are used in multiple places.

Appendix F

The Debugging Code Curry Implementation

```
module EclipseDebugWriter where

import Unsafe(unsafePerformIO)
import IO(Handle, hPutStrLn, hFlush)
import Socket
import Global(Global, GlobalSpec(..), global, readGlobal)

debugPort = 25025

debugSocketHandleGlobal :: Global Handle
debugSocketHandleGlobal = global
  (unsafePerformIO (connectToSocket "localhost" debugPort)) Temporary

-- message must not contain newlines
write :: String -> String -> IO ()
write logId message = do
  handle <- readGlobal debugSocketHandleGlobal
  -- if the user ignored the restriction, just send the first line.
  hPutStrLn handle $ logId ++ " " ++ (takeWhile (\x -> x /= '\n') message)
  hFlush handle

-- message must not contain newlines
genTrace :: (String -> IO ()) -> String -> a -> a
genTrace logWriter message x = unsafePerformIO $ do
  logWriter message
  return x
```

```
traceValue :: (String -> IO ()) -> a -> a
traceValue logWriter x = unsafePerformIO $ do
  logWriter (show x)
  return x
```

Appendix G

A Curry Module containing Test Cases for Type Errors

```
module TypeErrors where

-- contains some example functions having type errors.
-- purpose:
--   to test the type errors and inspect the exact
--   messages generated.
-- two sections:
--   1) some common functions used later
--   2) a set of outcommented sections, each containing one type error
--      if implemented

---------------------
-- HELPER FUNCTIONS --
---------------------

expectingInt :: Int -> Int
expectingInt x = x

expectingString :: String -> Int
expectingString = length

expectingTwice :: a -> a -> a
expectingTwice = const

exampleIntList :: [Int]
exampleIntList = [1,2,3,42]
```

```
exampleString = "Hello, World!"

applyFunction :: (a -> b) -> a -> b
applyFunction = id

------------------------
-- TYPE ERROR EXAMPLES --
------------------------

-- note that for certain cases, you will need to save the
-- document once and then modify it in some way to actually
-- get a type error message.

error1 :: Int
error1 = 'a'

error2 = expectingInt 'a'

error3 = expectingString exampleIntList

error4 = expectingString 'a'

error5 = expectingTwice 1 'a'

error6 = expectingTwice 'a' 1

error7 = if 'a' then 1 else 2

error8 :: Int -> Char
error8 x = id x

error9 :: Int
error9 = applyFunction (\x -> 'a')
```