CHALMERS
UNIVERSITY OF TECHNOLOGY

The CakeML Compiler Explorer

Visualizing how a verified compiler transforms expressions

Bachelor of Science thesis in Software Engineering

Rikard Hjort
Jakob Holmgren
Christian Persson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2017

The CakeML Compiler Explorer
Visualizing how a verified compiler transforms expressions

Rikard Hjort
Jakob Holmgren
Christian Persson

Supervisor:
Magnus Myreen, Department of Computer Science and Engineering.
Examiner:
Arne Linde, Department of Computer Science and Engineering.

Bachelor's Thesis 2017:24
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31 772 1000

Cover image:
Clker-Free-Vector-Images, 2014.
Available: `https://pixabay.com/en/tree-plant-dead-tree-silhouette-294114`.
Accessed: May 11, 2017.
Licence: CC0.

Typeset in LaTeX
Göteborg, Sweden, May 2017

Preface

This report is the result of a Bachelor thesis project at Chalmers University of Technology, conducted during the spring semester of 2017.

We would like to express our deepest gratitude to our supervisor, Magnus Myreen, for inspiring us with ideas and showing deep commitment to the project, spending countless hours on discussing issues small and large with us.

We would also like to thank the CakeML team, both for their support and quick answers to our questions, and for creating the interesting research project which is the CakeML compiler.

Finally, we would like to thank Bachelor thesis groups DATX02-17-03 and DATX02-17-29 for reading and giving us feedback on our report.

Rikard Hjort, Jakob Holmgren, and Christian Persson, Gothenburg, May 2017

Abstract

This report documents the development of a compiler explorer that provides insight to the inner workings of the CakeML compiler. The compiler explorer can interactively present information about an expression's origin and descent at different stages of compilation. The compiler explorer consists of a web application presenting the expression information and the CakeML compiler with our additions that enable the tracking of expressions. The CakeML compiler is developed in the HOL4 system; the web application user interface in React and the web server in PHP.

Getting insight into the inner workings of a compiler is difficult. Several tools exist for other compilers that either explain how a section of the source code relates to the compiled machine code or provide snapshots of different compiler phases. While these features are useful by themselves, combining them would give better insight into the compiler's transformations. The compiler explorer provides such a combination.

The gained insight provided by the compiler explorer can both help developers of the CakeML compiler find new optimizations and improve education about the compiler.

Keywords: compilers; education; visualization; ML; functional programming; logic programming; algorithms; formal languages

Sammandrag

Denna rapport beskriver utvecklingen av en kompilatorutforskare som ger insyn i CakeML-kompilatorns inre transformationer. Kompilatorutforskaren kan interaktivt presentera information om uttrycks ursprung efter olika kompilationssteg. Kompilatorutforskaren består av en webbapplikation som presenterar uttryck och deras härkomst, samt tillägg till CakeML-kompilatorn som tillåter denna att spåra uttrycks härkomst.

Att förstå hur en specifik kompilator fungerar är svårt. Det finns flera verktyg för andra kompilatorer som antingen visar hur källkod relaterar till maskinkod efter kompilering, eller kan visa ögonblickbilder efter olika kompileringssteg. Dessa funktioner är visserligen användbara var för sig, men att kombinera dem skulle göra kompilatorns transformationer mer lättförståeliga. Kompilatorutforskaren erbjuder en sådan kombination.

De insikter som kompilatornsutforskaren erbjuder kan hjälpa nuvarande utvecklare av CakeML-kompilatorn att identifiera möjliga kodoptimeringar samt underlätta utbildning om kompilatorn.

Nyckelord: kompilatorer; utbildning; visualisering; ML; funktionell programmering; logikprogrammering; algoritmer; formella språk

Contents

List of Figures

Nomenclature

AST	Abstract syntax tree, page 1
CIL	Compilation Intermediate Language, page 24
component	In the context of React: a function that takes a JavaScript object and produces a React element, page 11
element	In the context of React: a JavaScript object representing something that can be displayed on a web page, page 11
GUI	Graphical User Interface, page 11
HOL4	Higher Order Logic theorem prover, which CakeML is defined in, page 9
IL	Intermediate Language. Intermediate languages are used in compilers to transform a source program step-wise, by translating between similar but progressively more machine-like languages, page 5
orphan expression	An expression created directly from a declaration during a compiler pass, page 22
pass	Single traversal of the entire program by the compiler, page 1
PIL	, page 24
prelude	A large collection of predefined functions that the CakeML compiler adds automatically at compilation, page 13
prop	In the context of React: a parameter to a function that is a React component, page 11
trace	Data showing the path a piece of program in an IL has taken through the compiler, page 17

1
Introduction

Compilers play a central role in programming. They are the programs that can take source code written in a high-level language such as C, Java, Python, Haskell or ML and turn it into a machine-code file that a computer can run. In the process of compiling, many compilers optimize the code the programmer wrote to make the program more efficient while leaving behavior, or semantics, unchanged.

There is also a small set of compilers which are formally verified, meaning they are proven not to change the semantics of the input program. Examples of such compilers are CompCert which is a verified compiler for C code [1], [2] and CakeML which is likely to be the first verified compiler to be bootstrapped, i.e., that has been used to compile its source code [3]. The CakeML compiler is not only verified but also optimizes the compiled code [4, Sec. 5, 7.2] and by its design allows "implementation of optimisations at practically any level of abstraction" [4, Sec. 1].

The development team behind CakeML has expressed a desire to perform more optimizations. As an aid in this work, they have suggested a new tool, a *compiler explorer*, which would enable them to comprehend the inner workings of the compiler better. Such a tool should show several intermediate representations in the compiler side by side. Also, it should allow the user to select a piece of code in one representation and have the corresponding pieces of code in the other representations highlighted. The idea is that the new tool would enable the developers to easily inspect what the compiler is doing which would help in the development of optimizations. Furthermore, it is expected that such a tool would aid new developers in quickly gaining an understanding of the code. This report is about the initial development of such an explorer tool.

1.1 Problem specification

As it stands, the compiler has no debugging tool or other means for stepping through its code. Because of this, it is hard to exactly comprehend what the compiler does since all one can do is read its code and run sections of it manually, which is tedious. The result of this obstacle is that it is hard to identify possible code optimizations and to introduce the compiler to new developers.

A feature missing in the current compiler is the possibility to follow parts of the input program as it moves through the compiler. At the moment, the compiler creates *abstract syntax trees* (ASTs) which are traversed a large number of times. Each traversal called a compiler *phase*, or a compiler *pass*, introduces some change until the AST can be turned into machine code. In this way, the different parts of

the program are gradually transformed. However, there is no information relating a specific part of the program at a certain pass with its earlier or later forms.

1.2 Solution and contribution

We have created a way to follow expressions through the CakeML compiler by modifying the intermediate representations. We have also built a simple way to output intermediate representations from the compiler in a JSON format which can easily be consumed by a web application.

To show the usefulness of these new features we have built a proof-of-concept web application in which the user can compile a CakeML program, and for each intermediate representation present the information given by the compiler. By clicking an expression in one of these representations, the earlier and later forms of the same expression get highlighted.

This way to trace expressions through a compiler is, as far as we can determine, completely novel. Our way of encoding position information, which will be discussed in Section 4.1, is also quite minimal yet contains all information necessary to follow the transformations of an expression. Furthermore, our solution for outputting JSON information from the compiler is modular, and only small additions are required to add support for new compiler passes.

Finally, we have solved several issues around the presentation, such as how to follow expressions that are created by the compiler rather than appearing in the source code and how to present the internal states in a readable way even in the presence of De Bruijn indexing. De Bruijn indexing, which will be explained in Section 2.2.4, is a technique for binding variables in expressions that does not use any variable names, which makes the code hard to parse for a human reader.

A potential drawback of introducing these changes to the compiler is the overhead in computation and memory it carries with it. To tackle this, we have introduced a way of turning off the tracking data from the start, resulting in very little overhead.

1.3 Structure of this report

The structure of this report is as follows.

- In Chapter 2 we introduce technical knowledge needed to follow the remainder of the report. We give an introduction to verified compilation, CakeML and its internal structure; the HOL4 interactive theorem prover; and a framework for building a modern web application, React.
- After having introduced the necessary theory, we continue in Chapter 3 and define our goal and delimitations for the compiler explorer in detail. We also explain how we have worked, and examine projects with similar goals to ours.
- In Chapter 4 we describe our solution in detail, first focusing on the changes to the compiler and after that on how to use its output in a web application.

- Chapter 5 is a critical look at our own process and its possible effects on the final result. As well as the result itself, we discuss its fit as a solution to our problem, along with suggested future improvements.
- Chapter 6 is a closing summary of this report, its results, and our main discussion points.

We assume that the reader has a fundamental knowledge of functional programming or lambda calculus as this project relies heavily on these topics, especially the sections covering De Bruijn indexing and the changes we have made to the CakeML compiler. However, it is our belief that a reader unfamiliar with these topics but experienced in programming and familiar with the basics of context-free grammars should have little trouble following the main ideas presented.

2
Technical Background

In this chapter, we introduce important concepts that will play a crucial role in the rest of this report. Given the array of topics that needs to be introduced, this chapter is necessarily somewhat fragmented, and each section can be read without knowledge of the others. This siloing of concepts allows the reader to use this chapter as a general reference, and to look up concepts as needed. The structure of this chapter is as follows.

- Section 2.1 explains the concept of verified compilation and its importance in more detail.
- Section 2.2 covers the CakeML compiler, especially the concepts of intermediate languages, line annotations, and De Bruijn indexing.
- Section 2.3 gives an introduction to writing functions in HOL4.
- Section 2.4 covers React, a JavaScript library used to build our web application.

2.1 Verified compilation

The CakeML compiler is a verified compiler, meaning it is developed together with a mathematical proof which shows that the compiler preserves program behavior [3]. In this section, we briefly describe the idea of verified compilation.

While compilers play a crucial role in programming, they are themselves just programs, and thus susceptible to the same weaknesses as the programs they compile. Particularly in the case of an optimizing compiler, they might be large and highly complex programs. As such, they run the risk of having bugs. A bug in a compiler means that the compiled program behaves differently than what is specified in the semantics of the input language. Such a bug is called a *miscompilation*. The existence of miscompilations poses the question whether a given compiler actually can be trusted. If a compiler might miscompile, any proofs of the correctness of the program in the source language risk being void. And indeed miscompilation happens—compilers have been shown to have bugs [5].

Miscompilation is the motivating problem behind verifying a compiler. A verified compiler has been proven to never miscompile, i.e., never change the behavior of the compiled program. By using standard axioms and inference rules of logic, it can be shown that the compiler will not miscompile.

2.2 CakeML

This section describes some techniques used in the CakeML compiler that will be important to our project. Firstly, we give an overview of the CakeML compiler and its general structure. Secondly, we explain the role of some intermediate languages in the CakeML compiler. Thirdly, we explain De Bruijn indexing, a technique used in the compiler that our project needs to roll back to make the code human readable. Finally, we look at how the compiler creates line annotations for its input code, a feature which we will utilize later.

2.2.1 The compiler and its general structure

The CakeML compiler is a verified compiler for a significant subset of the Standard ML programming language and compiles to a wide range of popular architectures [4, Sec. 1]. A special type of semantics called functional big-step semantics are used to define program behavior [6], and it is proven that the machine code produced by the compiler has identical semantics to the input program [3]. The proof is conducted with the HOL4 theorem prover, which was in turn proven to be correct by Kumar et al. [7], and which will be presented in more detail in Section 2.3. Furthermore, the CakeML compiler can be run as a function in the HOL4 logic, which can be applied to the compiler source code, producing a machine code representation of itself that is verified; the compiler is thus bootstrapped and is likely the first verified compiler to be so [3].

In the process of compiling, the CakeML compiler uses 12 *intermediate languages* (ILs) on different levels of abstraction, from source to machine code. These languages and their roles in optimizing are described by Tan et al. [4].

Fig. 2.1 shows the progression of a source program through the compiler, with comments on each compiler pass. The compilation is shown from top to bottom, with each arrow representing a compiler pass, and each box representing an IL. In the code base for CakeML [8], each language has an abbreviated name relating to its role. For example, the sixth IL from the top, which introduces exhaustive pattern matches, is called `exhLang`.

2.2.2 The early intermediate languages

This report will deal mainly with the early intermediate languages, from the boxes "source AST" up to and including "no pat. match" in Figure 2.1. These languages all have roughly the same syntax. In this section, we will give an overview of what the syntax looks like to get familiar with its general structure.

Fig. 2.2 shows the syntax for the expressions in the source AST as an algebraic data type, simply called `ast` in CakeML. It captures all syntactically valid expressions in a CakeML program and is recursively defined. Every line after the first begins with a constructor, followed by one or more types, declaring what the type the arguments to the constructor must have. There are a few keywords in this definition that will recur in this report, which we will explain here. alist and list both indicate a list of a certain type, so string list indicates a list of strings. option

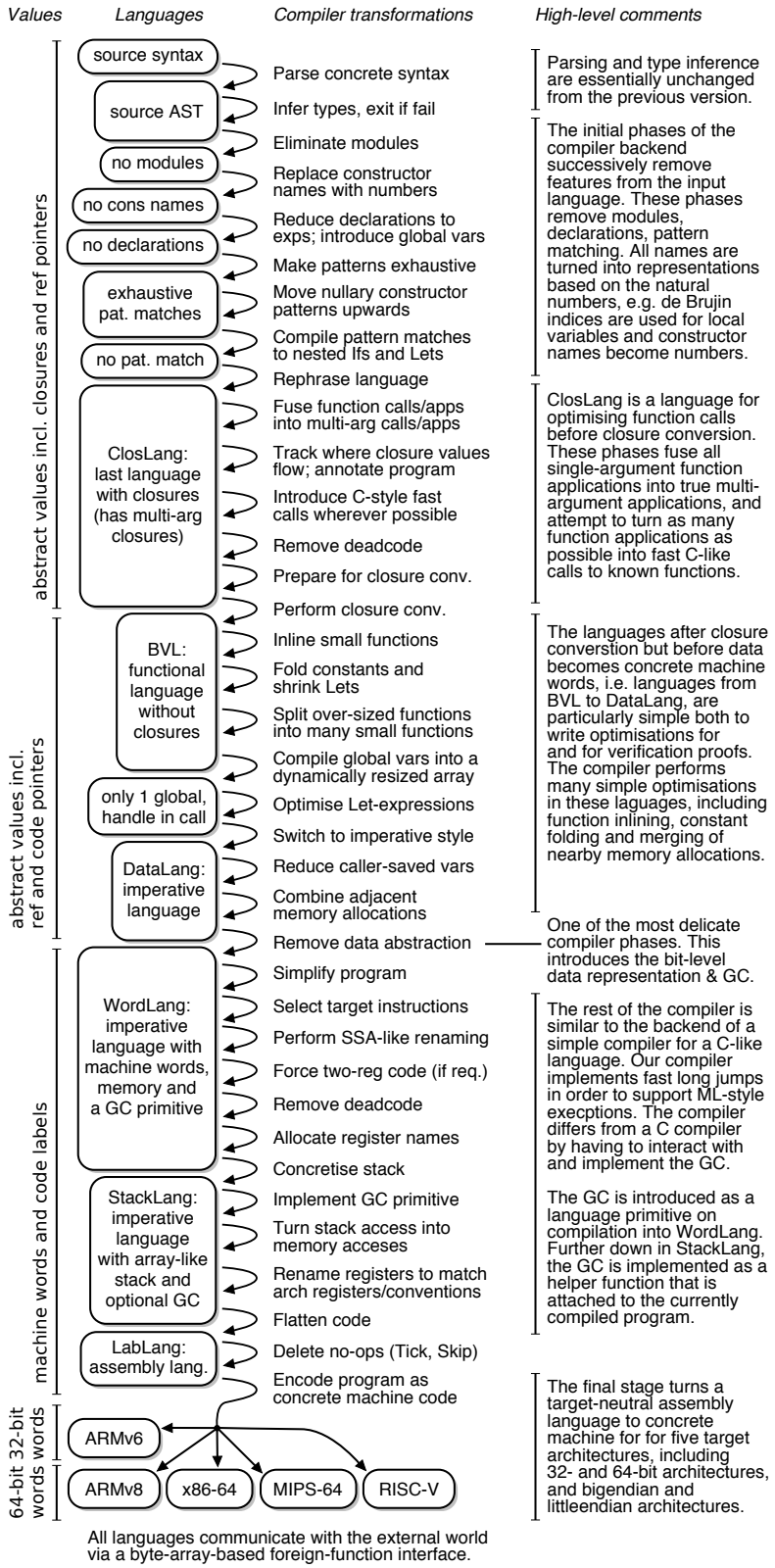

Fig. 2.1. Visual description of the CakeML compiler. Boxes represent intermediate languages (ILs) and arrows compiler passes. Illustration provided by Magnus Myreen [9]. Reproduced with permission.

```
exp =
    Raise exp
  | Handle exp ((pat, exp) alist)
  | Lit lit
  | Con ((string, string) id option) (exp list)
  | Var ((string, string) id)
  | Fun string exp
  | App ast$op (exp list)
  | Log lop exp exp
  | If exp exp exp
  | Mat exp ((pat, exp) alist)
  | Let (string option) exp exp
  | Letrec ((string, string × exp) alist) exp
  | Tannot exp t
  | Lannot exp locs
```

Fig. 2.2. `exp` data type in the `ast` language inside CakeML. `exp` represents general expressions in CakeML. This data type is representative for the expression data types in several of the languages that follow. `$` is used to give a fully qualified name to a type, so `ast$op` refers to the `op` type in the `ast` module. The × symbol indicates a tuple. Taken from the source code of the CakeML compiler [8].

means the value may but does not have to be present, so an expression of the type string option can be either NONE or SOME x where x is a string. id is defined as (ν, ξ) id = Short ξ | Long ν ((ν, ξ) id), simply meaning it is a Short of one type, wrapped an arbitrary amount of times by Long values with some other type. For example, (string, string)id can be used to define arbitrarily long qualified names, such as `foo.bar.baz`, which would be the value `baz` in the module `bar`, which in turn is inside the module `foo`.

The intermediate languages following `ast` are, in turn, `modLang`, `conLang`, `decLang`, `exhLang`, and `patLang`. There are of course more languages after these ones, but these are the ones this report covers. `ast`, `modLang` and `conLang` all have declarations, which disappear in `decLang`, from which point the program is represented by a single expression, which remains the case up until `closLang` which introduces a code table, i.e., an immutable code store.

Declarations in `ast`, `modLang` and `conLang` all look roughly the same. Fig. 2.3 shows the `dec` data type in `modLang`. A declaration in CakeML is the assignment of an expression to a variable, such as `val x = 3 + 5`, where `3 + 5` is the expression, and it would be encoded as a `Dlet`. There are some variants to this, such as declarations for mutually recursive declarations (`Dletrec`), but that is a minor point for the purposes of this report.

```
dec =
    Dlet num modLang$exp
  | Dletrec ((string, string × modLang$exp) alist)
  | Dtype (string list) type_def
  | Dexn (string list) string (t list)
```

Fig. 2.3. `dec` data type in the `modLang` language inside CakeML. `dec` represents general declarations in CakeML. This data type is also similar to the declaration data type in `conLang`. Top level declarations are encoded with Dlet, mutually recursive declarations are encoded with Dletrec, type declarations are encoded with Dtype, and new exceptions are encoded with Dexn. Taken from the source code of the CakeML compiler [8].

2.2.3 Line annotations on expressions

In the first compiler pass, from "source syntax" to "source AST" in Fig. 2.1, the parser phase of the CakeML compiler wraps each expression in the source AST in a line annotation, using the Lannot exp locs constructor. This constructor takes an expression to annotate and a pair of natural number pairs (called locs), representing the start and end positions of each expression. The first number pair is the line and column where the expression starts, and the second the line and column where it ends (inclusive, indexed from 1). If for example, the statement

```
val x = 3 + 5;
```

appears at the beginning of the first line, the starting position of the expression `3 + 5` is `(1,9)` and the end position is `(1,13)`. The fact that each location is unique—two expressions can not both start and end in the same place in the source text—will be useful as a point of departure for our tracing of expressions later, since we will need to assign unique identifiers to expressions. At the start of this project, the parser only annotated expressions and not declarations. However, similar annotations were added to declarations in April 2017[1].

2.2.4 De Bruijn indexing

Starting from `patLang`, represented by the seventh box in Fig. 2.1, the CakeML compiler uses De Bruijn indexing, a notation scheme originally invented for lambda expressions. This notation looks different from the standard variable naming used in lambda calculus as well as many high-level programming languages where a variable defined with the name `x` will from then on always be referenced by writing `x`. In this section, we give a short introduction to De Bruijn indexing.

De Bruijn indices are numbers which replace variable names according to a specific scheme. In short, De Bruijn indexing uses an ordinal number to indicate where a variable was bound [10]. Starting from 0, this means 0 represents the nearest

[1]The change was introduced by commit 7fbd1d01861385d3298aa5e6ad147a6de2b69f48 in [8].

Fig. 2.4. Graphical representation of De Bruijn indexing. The arrows relate each number to the λ that binds it. Adapted from [11], CC0.

enclosing lambda, 1 the next one, and so on. An example of a simple lambda expression can be seen in Expression 2.1, which contains four bound variables. With De Bruijn indices instead of variable names, we get the result seen in Expression 2.2. As can be seen by comparing the expressions, every instance of 0 denotes a different variable, namely their nearest enclosing λ. If there were any unbound variables in this expression, they would have an index larger than the number of lambdas enclosing it. For example, if any of the numbers were 3, it would reference an unbound variable.

$$\lambda x.\ (\lambda y.\ y\ (\lambda z.\ z))(\lambda u.\ x\ u) \tag{2.1}$$

$$\lambda\ (\lambda\ 0\ (\lambda\ 0))\ (\lambda\ 1\ 0) \tag{2.2}$$

Another view of this expression is shown in Fig. 2.4. Here Expression 2.2 is decorated with arrows from the numbers to the λ by which they are bound.

2.3 HOL4

CakeML is written using HOL4, the latest version of the HOL system which is designed to support interactive theorem proving in higher order logic, hence the acronym HOL. One interacts with the HOL4 system using the read-eval-print loop of the general-purpose programming language Standard ML [12].

In HOL one builds functions and theorems which describe them. Fig. 2.5 gives an example of a function in HOL. We deliberately use a contrived yet working example of a function that returns either its input or its input incremented by one. It does this by first calculating a local variable, *modRes*, and then pattern match on its value.

The resulting definition in HOL, seen in Fig. 2.6, looks only a little bit different. The result is a theorem stating that for all n's, inc_odd_num n will assume the value n + 1, n or ARB, where ARB represents an arbitrary number. The last case was not something included in the input but rather was inferred by HOL. In HOL, every function must be a total function. Since *modRes* has type `num` and we only covered the cases of 0 and 1, HOL adds a case for every remaining possible value.

```
val inc_odd_num_def = Define‘
  inc_odd_num n =
    let modRes = n MOD 2 in
     case modRes of
        | 1 => n+1
        | 0 => n‘;
```

Fig. 2.5. Example input code to HOL4. This code defines a simple function which increments its input by 1 if it is odd, and returns it unchanged if it is even.

```
inc_odd_num (n : num) =
(let (modRes : num) = n MOD (2 : num)
 in
  case modRes of
    (1 : num) ⇒ n + (1 : num)
  | (0 : num) ⇒ n
  | v ⇒ (ARB : num))
```

Fig. 2.6. HOL4 representation of the function defined in Fig. 2.5. The function is encoded as a theorem, and all possible cases for the result of *modRes* are covered. All variables are shown with their types. In this case, the only type is `num`, which represents natural numbers.

```
const Header = ({ firstName, lastName }) => (
<span>Hello, {firstName}! Your last name is {lastName}.</span>
)
```

Fig. 2.7. Example code for a simple React component. The text contained within the `<span>` tags is affected by the values of the `firstName` and `lastName` props, respectively.

2.4 React

For the graphical user interface (GUI) of our web application, we make use of a library called React, developed by Facebook [13]. The purpose of React is to alter a web page by changing its HTML dynamically.

Web browsers render web pages using the HTML language. HTML elements are organized together in a tree structure describing the structure and content of a web page. Web browsers parse this tree structure to display visual elements on the screen, and if the HTML elements in the tree change the web browser updates the visual elements.

React is based on a similar model to HTML, creating a tree of React *elements*. React elements are JavaScript objects that represent something that can be rendered by a web browser. In a web application using React, the React framework converts React elements to HTML elements, which are then rendered by the web browser [14].

Instead of manipulating React elements directly, one can define React *components*, which are JavaScript functions that return React elements [15]. The parameters to a React component are called that component's *props*. Since a React component is an ordinary JavaScript function, the props can be of any type.

Since React components return React elements, they can be used in places where React elements are expected. Thus, one can denote components as HTML tags, i.e. the component `Foo` can be denoted as `<Foo>`. This convention will be used throughout this report.

A code example of a simple React component representing a web page header can be seen in Fig. 2.7. The `<Header>` component takes two props, `firstName` and `lastName`. It returns a React element representing a HTML `<span>` element containing the text "Hello, $firstName$! Your last name is $lastName$." where $firstName$ and $lastName$ are substituted with the values of the `firstName` and `lastName` props, respectively.

3
Prestudy

In this chapter we take a closer look at the problem and our suggested solution, before going into the solution in depth in Chapter 4.

In Section 3.1 we define our main goal for the project, which is to build a web application that allows the user to click expressions and see their ancestors and descendants in other ILs. To do this, we need to introduce changes to the compiler, both for following expressions through the compiler and for outputting the necessary data from the compiler. In Section 3.2 we go on to define the outer boundaries of this project, i.e., some things that are of interest but that we will not concern ourselves with as part of this project.

We also describe the way in which we go about building the solution. Firstly, we explain how we collect user requirements from the core developer team in Section 3.3. Secondly, we explain how we divide our main goal into high-level subgoals that can be completed separately in Section 3.4.

We conclude this chapter by surveying the field of similar projects in Section 3.5, and conclude that we find no project that provides the feature of following expressions as our solution does.

3.1 Goal for the final product

The final product that this project sets out to produce is called the "compiler explorer". We based our goal for this product on correspondence and discussions with the CakeML development team.

The compiler explorer should be a web application in which the user can write CakeML source code in an editor or text field, compile it and get a printed view of what the program looks like after each compiler pass. Furthermore, the user should be able to view the state at two differents points in the compilation side by side, click a declaration or expression in one of these states, and have the corresponding parts of the program in the other state highlighted. It should be possible to view the initial state (the source program) in this way, including highlights.

The purpose of the compiler explorer is twofold. Firstly, it would allow for current developers to more easily understand the intricacies of the compiler passes, the main purpose being that they can then identify potential optimizations the compiler could perform. Secondly, it would make it easier for new CakeML developers to learn and understand the compiler.

Given that we need information from the compiler before we can build the web application, we decided to gear the project primarily towards figuring out the best

way of modifying the existing compiler, verify the soundness of our suggested modifications with the developer team, and then making the necessary changes.

Finally, we set a goal for this project as a whole of developing a basic understanding of the CakeML compiler, which we believe will give us a better understanding of compilers in general, and of the process necessary for formally verifying a compiler.

3.2 Delimitations in scope

Due to the time constraints and small workforce in this project, we have deemed it necessary to limit our scope. The following sections will make explicit some aspects of the compiler explorer that are important for its usability and quality, but which we have chosen to not focus on.

3.2.1 Speed and responsiveness of the web application

The compiler explorer makes use of tree traversal algorithms for highlighting. These traversal algorithms visit every node in quite large ASTs, making the compiler explorer somewhat slow. The highlighting can take tens of seconds in the Chromium browser, running on modern laptops. Several possible optimizations could solve this problem, and we discuss them in Section 5.3.2. However, in the face of other priorities, such as having an otherwise complete user interface, we have opted not to perform these optimizations.

3.2.2 Tracing source position of declarations

As mentioned in Section 2.2.3, declarations did not get annotated with their position in the source code until April 2017, several months into this project. We considered a workaround but ultimately decided it would be too complex to figure out source position of declarations. Therefore, we have opted to leave out highlighting for declarations.

3.2.3 Tracing prelude code

After reading and parsing the source program as input, the CakeML compiler adds a *prelude* to the source program, which contains a large number of predefined functions. This prelude is not at any point during the rest of the compilation separated from the user-supplied source code. However, it contains no position information. Therefore, this code appears in the compiler explorer, but without the possibility to follow its transformations, and the user of the web application will have to search through it to find the parts of the program that was written by them. We have discussed several methods for marking and handling the issues with the prelude code but ultimately decided not to implement a solution. However, we suggest it as future work and outline a solution in Section 5.6.

3.2.4 Updating proofs

After our modifications to the compiler, some proofs for the compiler need to be updated. After consulting with our supervisor, we have decided to leave the updating of proofs to him and the other developers, since to update the proofs we would have to learn much more about theorem proving in HOL than there has been time for.

3.3 Collecting user requirements

The way we collect user requirements and clarify specifications is by direct contact with the CakeML developers. Our main contact with the team is through our supervisor, Magnus Myreen, who is one of the current developers listed on the CakeML project website [16]. We also participated in an initial conference call with the developer team at the start of the project, and have posted questions on the CakeML developer mailing list and the CakeML IRC channel, all of which are referenced from the CakeML website, mentioned above.

The main requirements for the compiler explorer, such as that it should be a web application, print ILs and have the described highlighting feature, have been solicited from our supervisor and were defined in the project specification from the start. The same is true of the purpose of the compiler explorer in aiding optimization efforts and learning. An informal survey conducted on the developer mailing list in May 2017 indicated that the other developers had similar expectations on the explorer. The survey and responses can be found in Appendix A.

3.4 Subgoals of project

We turn our attention to the division of this project into discrete, high-level subgoals. As the web interface will need to have information about the state at every IL, the compiler will have to provide such information, meaning that it must both contain the information and output it at compilation. Based on this simple analysis, we have arrived at the following subdivision, which we cover in the following three subsections. In reaching these goals, it is also an objective of ours to produce a high-quality solution from a software development perspective.

3.4.1 Adding position information to expressions

To be able to follow an expression from source code to any intermediate language, ancestry needs to be encoded in the expressions somehow. From looking at the expressions in one IL, it should be possible to apply an algorithm which decides what expressions in a previous IL it has originated from, and what expressions in a later IL it gives rise to. It is worth noting that since the CakeML compiler lives within HOL, the solution must be purely functional and have minimal impact on existing proofs, i.e., the compiler is not allowed to change much in structure.

3.4.2 Outputting position information from the compiler

Once we have found a way to structure positional information and track it through the ILs, we will need to output the state of the program, including position information, at every IL. The output will need to be structured in such a way that the web application can easily consume it. A suitable format is thus JSON since it is a standard language that JavaScript can easily read in.

3.4.3 Building a web application

We will need to create a web application which can present the information we output from the compiler. The web application should match the description in Section 3.1. Solving this problem also includes figuring out an intuitive way to represent the ILs textually. While there are no clear rules for how to represent the ILs as text, using standard conventions for functional programming and context-free grammars we should find ways of printing the trees in a readable fashion so that it is intelligible to most programmers.

3.5 Similar projects

To the best of our knowledge, there exists no tool for visualizing the transformations of a compiler in the way we are proposing, where each IL can be viewed, and corresponding expressions in different ILs are highlighted together. In our search for related work, we have come across four projects which each provide parts of what we are proposing.

3.5.1 The nanopass compiler framework

The nanopass compiler framework by Sarkar et al. provides an unparser:

> The unparser can also translate the record structures into their implied parenthesized forms, i.e., with no host-language translations, allowing the student to pretty-print intermediate language code. [17, Sec. 4.1]

However, only this possibility to pretty-print is mentioned, and no mention of any tracing capabilities is made.

3.5.2 LLVM Visualization tool

The LLVM project has a visualization tool which, like the nanopass framework and our explorer, can show the state of the compiler after each pass [18]. The visualization tool offers several views of the state after each pass, showing call hierarchies, unparsed code with syntax highlighted, control flow graphs and more, but there is no mention of any tracing capabilities here either.

3.5.3 Godbolt's compiler explorer

There exists a tool that is also called "Compiler Explorer", which supports the languages C++, Go, D, and Rust [19]. For clarity, we will refer to this tool as "Godbolt's Compiler Explorer", after one of its maintainers. Godbolt's Compiler Explorer lets the user edit the source code, and get a line-by-line view of how the resulting assembly code relates to the source code. One feature that is missing from Godbolt's Compiler Explorer, however, is the capability of viewing the intermediate code after different compiler passes.

3.5.4 CakeML's old compiler explorer

There exists a web application for exploring CakeML, also called "Compiler Explorer". For clarity, we will refer to this web application as the "old explorer". The old explorer's web page states that one can "Write CakeML code and see how it is transformed by each phase of compilation" [20]. When a program is put into the old explorer, the output is the resulting ASTs for some ILs. However, this implementation lacks the desired feature of allowing the user to highlight parts of the program and see what parts of the program the highlighted part corresponds to in later or earlier ILs.

4
Results

In the course of this project, we have made changes to the CakeML compiler, adding the possibility to compile the input code into a JSON object which shows the state of the compiler after each pass, together with information about which parts of each intermediate AST correspond to each other. These changes have been made in a fork of the main CakeML project [21].

With the help of these changes to the compiler, we have created a proof-of-concept web application which can compile the source code and output the compiler state at several ILs [22]. The user may click an expression in an IL and have the corresponding expressions highlighted in both previous and later ILs, thereby seeing the ancestry of each expression, so as to better understand the compiler's inner workings.

These results were achieved by three technical contributions, two of which are modifications to the CakeML compiler, and one which is a web application with algorithms for determining ancestry.

- The first contribution, covered in Section 4.1, is a new data type, the *trace*, which holds data of which path a section of the input program has taken through the compiler.
- The second contribution, covered in Section 4.2, is three new ILs which are used to compile existing ILs into JSON format.
- The third contribution, covered in Section 4.3 is the web application as mentioned earlier, which has a React [13] GUI and contains algorithms for traversing the JSON given from the compiler to show ancestry of expressions.

4.1 Adding traces to the compiler

A central problem is to come up with a way to track the path a piece of the input program takes through the compiler. If we imagine a program moving through the compiler, from source code to binary format, then at every step we want to ask where each expression in the current state came from in the previous state, the one before that and so on.

Our contribution that solves this problem is the trace data type, a piece of data that annotates each expression in each IL with information of what path that expression has taken from the source code input up to that point. A trace essentially conveys, in a minimal form, the following things: an expression's exact position

```
tra = Empty | None | (▷) tra num | Union tra tra
```

Fig. 4.1. `tra` data type. Used for encoding the path any one expression has taken through the compiler, where ▷ (Cons in prefix form) encodes traces being split and Union encodes expressions being merged.

in the source code, where that expression has been split into one or more other expressions and where two or more expressions have been joined into one. This section explains the structure and use of traces, and how they have been added to the CakeML compiler.

4.1.1 The `tra` data type

The traces are implemented with the `tra` data type. A `tra` value is intended to be added to the information contained in an expression. For example, if the compiler previously contained an expression constructor such as

```
| Let (string option) ast$exp ast$exp
```

then after our modifications, the constructor would instead be

```
| Let tra (string option) ast$exp ast$exp
```

where a `tra` has been inserted right after the constructor name.

Fig. 4.1 shows the definition of the `tra` data type. It is a recursive type with two non-recursive constructors, Empty and None, and two recursive ones, ▷ and Union. ▷ is an infix version of a constructor named Cons, in the tradition of how the word is used in LISP, where "compositions of cons form expressions of a given structure out of parts" [23, Sec. 3]—essentially a device for building recursive structures, for example, lists—and we will use it both as Cons in prefix form and ▷ in infix form in this report.

4.1.2 Encoding ancestry with traces

This section describes how traces are used in the compiler explorer. We will first go over how traces are most often instantiated, and after that the different ways, they may grow. From this, we can explain how they can be checked to find ancestors and descendants of a given expression.

At the outset, every expression in the source AST is wrapped in a Lannot constructor, as described in Section 2.2.3. In the first IL after the source AST, `modLang`, these line annotations are converted into `tra` values. Each line annotation consists of four numbers indicating the position of the expression in the source code: the start row, start column, end row, and end column. These numbers are put into four Cons constructors, mimicking a list of four elements. Fig. 4.2 shows how this is done in the conversion from source AST to `modLang`. When a Lannot is encountered, the start and end positions (*st* and *en*) are read and turned into a trace, which is then passed

```
compile_exp t env (Lannot e (st,en)) =
(let t' =
     if t = None then t
     else (((Empty ▷ st.row) ▷ st.col) ▷ en.row) ▷ en.col
 in
  compile_exp t' env e)

compile_exp t env (App op es) = App t op (compile_exps t env es)
```

Fig. 4.2. Converting from source AST to `modLang`. The Lannot case shows how a line annotation is converted into a trace that is passed along to the next call of compile_exp. The App is an example of how this trace is attached to a modLang expression.

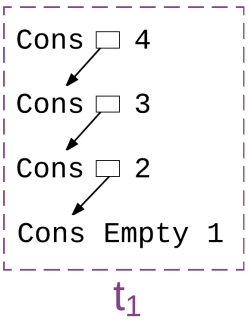

Fig. 4.3. Structure of the first trace t_1 of an expression. Expression traces are created with four initial numbers indicating the position in the source code, in this example starting on row 1, column 2, and ending in row 3, column 4.

along to the next call of compile_exp. The only exception is if the trace None has been passed in, the motivation for this is explained in Section 4.1.5. For expressions other than Lannot, the trace that is being passed along in the function call gets attached to the created `modLang` expression. Since every expression is wrapped in a line annotation, this means each expression will have its source position encoded in its trace.

Fig. 4.3 shows what the structure of a trace t_1 would be for an expression starting in row 1, column 2, and ending in row 3, column 4, it is essentially a list of the four initial numbers. By including this information in the trace, it will be possible to locate and highlight the position or positions in the source code where an expression in any IL originated. This way of constructing traces also ensures that every trace is unique at the start since no two expressions could start and end in the same place in the source code[1]. The way we designed traces will be discussed in Section 5.2.1.

For all compiler passes following the initial one, traces are only allowed to grow or remain unchanged, never shrink. Traces grow either by being extended with another Cons or by joining two traces together with a Union. It is also possible for a trace

[1]Note, however, that expressions can contain other expressions.

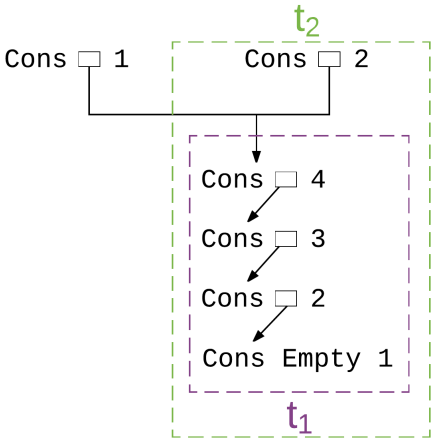

Fig. 4.4. Trace t_1 being split into two traces using Cons. Both Cons values in the top of the figure are new traces, but they wrap the same base.

```
compile_exp bvs (Handle t e1 pes) =
Handle (t § 1) (compile_exp bvs e1)
 (compile_pes (t § 2) (NONE::bvs) pes)
```

Fig. 4.5. Conversion of Handle from `exhLang` to `patLang`. The conversion includes the splitting of a trace. The § symbol is an inline version of mk_cons, covered in Section 4.1.5, and can for now be thought of as being exactly like ▷. Taken from the source code of our modified version of the CakeML compiler [21].

to remain unchanged from one pass to the next, and this is, in fact, most often the case. Only when an expression get split into several ones, or two expressions get combined into one new, extended traces get created for the new constructs.

Fig. 4.4 shows what happens to a trace t_1 in a situation where the expression carrying it gets split into two expressions. A split happens every time a trace occurs more than once on the right-hand side of a definition, such as in the case of Fig. 4.5, which shows part of a function taken from the conversion of `exhLang` to `patLang`. The transformation is from a Handle constructor in `exhLang` to a Handle constructor in `patLang`. Here the newly created Handle value is assigned a new trace, wrapped in an additional Cons, and another newly created trace is passed to the compile_pes function, which compiles patterns. Each new trace gets wrapped in a Cons with a natural number, such that the number is different for each of the newly created traces. As a general rule, the numbers start at 1 and are incremented at each occurrence on the line, reading left to right.

Fig. 4.6 shows the situation where instead of two or more traces occurring on the right-hand side of a transformation, there are two or more traces on the left-hand side. This indicates several expressions are merging, and the Union constructor is necessary. In Fig. 4.6, the trace t_2 seen in Fig. 4.4 gets merged with a different trace altogether.

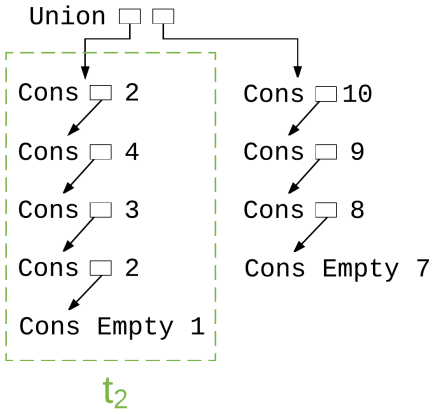

Fig. 4.6. Two traces being merged into one using Union. The first trace t_2 being merged was created by the process described in Fig. 4.4 and the second trace was created directly from a line annotation, as in Fig. 4.3.

4.1.3 Decoding ancestry from traces

Because of how traces are designed, it is a simple matter to find the ancestors and descendants of an expression based on its trace. This is because 1) an existing trace can only be expanded at each new compiler pass, and 2) all traces are unique from the start. However, it is not immediately obvious that this must work.

An expression e_1 with the trace t_1 is the ancestor of another expression e_2 with the trace t_2, if and only if t_2 can be derived from t_1. For a familiar metaphor, we can view the traces as binary trees, where a Cons constructor is a node with a numerical value and a left child, Union is a node with no value and two children, and Empty is a leaf node with no value. Then we can say with certainty that the second expression was derived from the first if t_1 is a subtree of t_2.

A more formal algorithm for determining ancestry is given in Fig. 4.7. It is presented here in imperative pseudo-code rather than JavaScript, which is the language used for this algorithm in the web application.

1) If either $t(e_1)$ or $t(e_2)$ is Empty, but not both, return FALSE.

2) If $t(e_1) = t(e_2)$, meaning the traces are identical, return TRUE.

3) If $t(e_1) =$ Cons t_c n, set $t(e_1) = t_c$ and start from 2).

4) If $t(e_1) =$ Union t_1 t_2, then

   (a) set $t(e_1) = t1$, repeat from 2) and save the result; then

   (b) set $t(e_1) = t2$, repeat from 2) and save the result; then

   (c) return the conjunction of the results.

Fig. 4.7. Algorithm for determining ancestry. Let $t(x)$ be the trace of expression x. Let e_1 and e_2 be expressions. Then e_2 is an ancestor of e_1 if the algorithm returns TRUE.

This method is certain to work thanks to how we grow traces. At the outset, all expressions have unique traces (we will take unique to mean that it can not appear twice in the same IL), since they come from unique source positions. A trace may be split by being wrapped in a Cons, encoding that the new expression is a descendant of the old one. For example, the trace in Fig. 4.3 is an ancestor trace of both new traces in Fig. 4.4, and is also a subtree of both of them, and this is the only way in which these new traces could have come about. Furthermore, both new traces are unique, so they, in turn, will safely match only to their descendants. If two traces are merged into a Union, then both these traces will be matched as ancestors to the Union trace. The Union trace will be unique since it is guaranteed that both merged traces were unique.

4.1.4 Adding traces to declarations

There is an important exception to the rule of expressions having traces that start with their source position, and that is the expressions which are created directly from declarations. For example, in the conversion to `decLang`, the first IL without declarations, all declarations are turned into expressions. These new expressions have no incoming trace data in them since declarations do not contain traces. Because of this, we face the challenge of coming up with a way of instantiating traces that is always globally unique. That is, the new traces must both be unique at the outset, and they may never accidentally grow into another identical trace.

The solution to this problem is a special trace for *orphan expressions* for each IL. Fig. 4.8 shows such a special trace for the conversion from `conLang` to `decLang`, the special trace is called od_tra for "orphan `decLang` trace"[2]. The structure of this trace is that of a standard orphan trace, the Union Empty Empty construct which is guaranteed to never conflict with a non-orphan trace, wrapped in a Cons. The

[2]In "od_tra", "o" is for orphan and "d" is for `decLang`.

```
orphan_trace = Union Empty Empty
od_tra = (orphan_trace ▷ 3)
```

Fig. 4.8. Start trace for orphan expressions in `decLang`. The construction Union Empty Empty could never appear at the start of a non-orphan expression, as they all start with a series of Cons values. By wrapping the basic orphan_trace in an additional Cons with a number unique for each IL, we obtain a basis for adding traces to orphan expressions.

Cons is needed to distinguish orphan traces in different ILs. In this case, the number provided to Cons is 3, since `decLang` is the third IL in the compiler backend[3].

The special orphan trace is added to each expression that is created directly from a declaration, after first being wrapped in an additional Cons with a unique number. The first orphan expression to be created gets the number 1, the second one the number 2, and so on. This number has to be explicitly passed from function to function and get incremented on the way, which is a somewhat intricate process which we shall not cover in detail here. The curious reader may, for example, examine the use of the functions compile, compile_prog, compile_prompt and compile_dec in the file `con_to_decScript.sml` in the source code of our modified version of the compiler [21] to learn how the unique numbering is achieved.

4.1.5 Turning traces off using None

The only constructor of `tra` we have not covered so far is None. This constructor serves a special purpose, namely turning traces off in an efficient manner, so as to avoid the computations and memory usage necessary for using traces. By passing along a None as a trace, no trace data will be accumulated in the compilation process. The reason is that Cons or Union are never used directly in building traces, except for when creating the initial trace as seen in Fig. 4.2. However, if a None is seen here, then a None gets passed on in the recursive call. In all other cases where traces are built, the smart constructors mk_cons and mk_union are used. Fig. 4.9 and Fig 4.10 show these constructors. Looking at mk_cons in Fig. 4.9, we can see that it takes the same arguments as the Cons constructor, but also checks whether the given trace is None. If the given trace is None, mk_cons returns None as well. Otherwise, it simply creates the corresponding Cons value. mk_union, as seen in Fig. 4.10, behaves similarly, and return None if one of the two given traces is None. By passing a None value as a starting value to the compiler, i.e., to the compile_exp function shown in Fig. 4.2, no actual trace is ever created, and all expressions simply get annotated with None.

[3] A similar construct used in `modLang` uses the number 1 since `modLang` is the first IL.

```
tr § n =
case tr of
  None ⇒ None
| _ ⇒ tr ▷ n
```

Fig. 4.9. mk_cons, as its infix version §. This function is used instead of calling ▷ directly. It checks whether the given trace is None and if so returns None. Otherwise, it works exactly like ▷.

```
mk_union tr1 tr2 =
case tr1 of
  None ⇒ None
| _ ⇒
    (case tr2 of
       None ⇒ None
     | _ ⇒ Union tr1 tr2
```

Fig. 4.10. mk_union. This function is used instead of calling Union directly. If any of the traces given to mk_union is None, then it returns None. Otherwise, it works exactly like Union.

4.2 Intermediate languages for output

Keeping with the convention of converting a program from one format to another by making one or more passes which convert the program from one IL to the next, we modified the compiler by adding three new ILs. These ILs help with the conversion of an IL used in compilation to a machine-readable output format which can be used to display that language. Since these ILs are only used for presentation purposes, and not for executing, they have only syntax and no defined semantics. To disambiguate the use of the term "IL", the terms "CIL" and "PIL"will be used in what follows. CIL shall refer to the ILs for compilation which are existing in the CakeML compiler, such as `modLang` or `patLang`. PIL shall refer to the ILs for presentation, which we have defined in the course of this project.

To output the compiler state at any CIL in a format that can be used to present the CIL in an external application, three passes are made, with a conversion to a new PIL at every pass. The first of the PILs is called `presLang` and represents anything that can be presented. The second PIL is called `displayLang` and structures the output in a way which defines how it should be displayed. The third PIL is called `jsonLang` and has the structure of a JSON object, which can then be converted to a string which can be pretty printed by the compiler. Fig. 4.11 shows the flow between CILs and PILs in the compiler, where horizontal arrows represent compilation through CILs with machine code as the target language, and vertical arrows represent compilation through PILs with JSON as the target language. The dashed

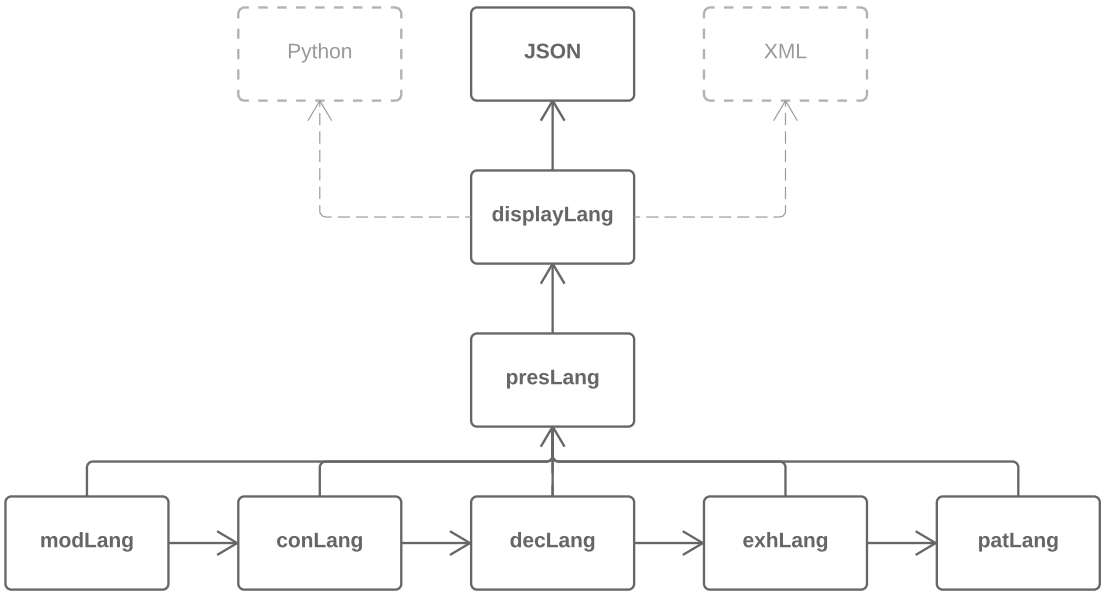

Fig. 4.11. Flow of a program through the modified compiler. Horizontal arrows indicate compilation with machine code as the target language, and vertical arrows represent compilation with JSON as the target language. The dashed boxes and arrows indicate possible alternative output formats to JSON.

boxes represent other possible target languages, to emphasize that outputting JSON is specific to our purposes of making a web interface, but another application might have better use of some other target language of the explorer. We now turn to describing each of the PILs more closely.

4.2.1 `presLang`

The first PIL is `presLang`, which encompasses anything that can be presented. `presLang` has an expression type with one constructor corresponding to every constructor in every CIL we have present in the compiler explorer. This includes declarations, patterns, and expressions.

Up to and including `exhLang`, each CIL has each of its constructors mirrored exactly in `presLang`. Most of these CILs have very similar syntax, so the same constructor in `presLang` can represent constructors in several of these CILs. When a constructor has had the same name but taken different arguments in different CILs, we have created a new data type to solve the problem. An example of this, for the constructor `Con`, can be seen in Fig. 4.12, where the `conF` datatype represents the `Con` constructor for `modLang`, `conLang`, and `exhLang`, respectively.

From `patLang` and onward, there is no longer an exact correspondence between `presLang` constructors and the constructors in the CIL, due to the encoding of De Bruijn indices, which will be explained further in Section 4.2.4.

Every CIL needs its own function to be converted to `presLang`, partly since otherwise types will not match. This is represented in Fig. 4.11 by the arrows from each CIL to `presLang`.

```
conF =
  Modlang_con ((string, string) id option)
| Conlang_con ((num × tid_or_exn) option)
| Exhlang_con num
```

Fig. 4.12. `conF` data type in `presLang`. The data type shows how `presLang` handles that the `Con` constructor differs in the CILs. The cross represents the tuple type.

```
sExp =
  Item (tra option) string (sExp list)
| Tuple (sExp list)
| List (sExp list)
```

Fig. 4.13. `sExp` data type in `displayLang`. This data type is used to structure expression in a general way, so that they may be easily translated to the desired output format.

4.2.2 `displayLang`

The second PIL is `displayLang` with the purpose of structuring any `presLang` expression in such a way that it, after restructuring, corresponds to how each constructor is to be represented in textual output.

The constructors of `presLang` are transformed into expressions of the form showed in Fig. 4.13. Looking at the three constructors, it is clear that the structure is far more basic than that of `presLang`. Any expression can be seen as a tree with three types of nodes: Tuple and List nodes which contain no data other than it being a Tuple or List, respectively, and Item nodes with a name in the form of a string, and optionally a trace. This structure can easily be captured in a programming language by using only dictionaries, tuples and lists. If the language lacks tuples, they can be encoded by lists and some signifier whether a list should be interpreted as a tuple or not, such as a boolean value. For our purposes, we will use a JSON format, as we will see in Section 4.2.3, but these structures could easily be captured in Python objects, an XML schema, S-expressions or some other suitable data format.

We have chosen to let the trace data type appear in the `displayLang` as is. That is, we have not encoded traces into `displayLang` expressions, which we could have done since traces are simply trees. However, since the trace plays a central role in the usage of the explorer data, we consider it valuable to be able to treat traces separately in the final output, which would be hard if the trace data was mixed in with expression data.

As can be seen in Fig. 4.11 by looking at the connection between `presLang` and `displayLang`, a single pass is needed and does not hinge upon which CIL is

```
obj =
    Object ((string, obj) alist)
  | Array (obj list)
  | String string
  | Int int
  | Bool bool
  | Null
```

Fig. 4.14. `obj` data type in `jsonLang`. This data type faithfully mimicks the grammar of JSON values as specified in [24], with the exception of not allowing floating point numbers.

represented in `presLang`. Thus a single function is needed, which can convert any `presLang` constructor into the `displayLang` format.

The only type of data that can be expressed in the tree is strings and traces. As such, all information about a `presLang` expression, except for its traces, must be encoded as strings. This highlights the purpose of `displayLang` that it is meant only to display, and to convert an expression to `displayLang` is to make explicit some desire on how it should be displayed, in string format. Even numbers need to be encoded as strings, since the HOL `num` datatype allows arbitrarily large integers, which could not be fully captured in JavaScript numbers, or any other language that uses a fixed number of bits for number representation.

4.2.3 `jsonLang`

The final PIL in the chain between a CIL and output is `jsonLang`. The structure of `jsonLang` can be seen in Fig. 4.14, which shows the `obj` datatype. The grammar of `obj` conforms to the specified structure of JSON objects [24]. With objects being defined as a sequence of members, each being a string (the name) and a value (which can be any other type of JSON value), arrays as sequences of JSON values, and bare the types of bare values being strings, numbers, booleans or the `null` value. The only deviation from the JSON specification is that this grammar does not support floating point numbers, only integers. The reason for this is that CakeML only supports integral numbers [25, Line 71].

Fig. 4.15 shows the function converting from `displayLang` to `jsonLang`. It encodes any Item as an object with the properties name and args, where name gets the value of the string in the Item, and the args the value of the list of arguments, converted to a `jsonLang` array. The trace is added as a property if it exists. Further, the function encodes a Tuple as an Object with a boolean indicating it being a tuple and an array of the elements. A List is simply encoded as an array.

As can be seen in the clause in Fig. 4.15 that converts Item objects, traces are converted to `jsonLang` by a trace_to_json function. This function is shown in Fig. 4.16. It encodes a trace as an Object with a name property, detailing which constructor is used, and other fields dependent on the constructor.

```
display_to_json (Item tra name es) =
(let es' = MAP (λ a. display_to_json a) es in
 let props = [("name",String name); ("args",Array es')] in
 let props' =
     case tra of
       NONE ⇒ props
     | SOME t ⇒ ("trace",trace_to_json t)::props
 in
   Object props')
display_to_json (Tuple es) =
(let es' = MAP (λ a. display_to_json a) es
 in
   Object [("isTuple",Bool T); ("elements",Array es')])
display_to_json (List es) =
Array (MAP (λ a. display_to_json a) es)
```

Fig. 4.15. Translation of `displayLang` to `jsonLang`. This HOL theorem describes how to interpret a `sExp` value from `displayLang` as a `obj` value, so as to structure it into the form of a JSON object.

```
trace_to_json (tra ▷ num) =
Object
  [("name",String "Cons"); ("num",num_to_json num);
   ("trace",trace_to_json tra)]
trace_to_json (Union tra1 tra2) =
Object
  [("name",String "Union"); ("trace1",trace_to_json tra1);
   ("trace2",trace_to_json tra2)]
trace_to_json Empty = Object [("name",String "Empty")]
trace_to_json None = Null
```

Fig. 4.16. Translation of trace to `jsonLang`. This is a HOL theorem that when applied equates every Cons (▷) value to an Object of a certain structure, the same for every Union value, and so on.

```
pat_to_pres_exp h (Var_local t var_index) =
Var_local t (num_to_varn (h − var_index − 1))

pat_to_pres_exp h (Fun t e) =
Fun t (num_to_varn h) (pat_to_pres_exp (h + 1) e)
```

Fig. 4.17. Removing De Bruijn indexes in conversion to `presLang`. The function pat_to_pres_exp, of which two clauses are shown here, takes a number, h, in addition to a `patLang` expression. h is the number of currently bound variables in the scope, which is used to assign names to variables. The function produces a `presLang` expression, where variable names are re-introduced. For example, a Fun expression in `patLang` has no variable name, but a Fun in `presLang` does.

In the end, the `jsonLang` object is transformed into a string which the compiler backend can output, and which can be consumed by the web application.

4.2.4 Handling De Bruijn indices

In `patLang`, De Bruijn indices are used in place of variable names, but when presenting expressions in the explorer from `patLang` the indices are replaced by variable names. To achieve this, we reverse the De Bruijn indexing process when converting from `patLang` to `presLang`.

As the variable names have been discarded by the compiler, we invent new names. The outermost variable is given the name "a", the second outermost variable given the name "b" and so forth. When "z" has been bound, the next binding inside that scope will bind "aa", the next one "ab" and so on. Thus, by looking at a variable name binding, one can determine how many variables are bound in the current scope.

In CakeML, De Bruijn indices are more sophisticated than in lambda calculus, as there is more than one way to bind a variable and a richer syntax. Fig. 4.17 shows parts of the function that converts an expression from `patLang` to `presLang`. The first case shows how a variable is given a name as described in the previous paragraph. The function num_to_varn returns "a" when given 0 as input, "b" when given 1, and so on. By taking $h - num - 1$, num being the De Bruijn index of the variable, we obtain which order the current variable has, i.e., 0 for the outermost variable, 1 for the second outermost variable, and so on. The second case shows how h is increased when a binding is encountered. h is used to calculate what name the new variable should get, and pat_to_pres_exp is called recursively, but with h incremented.

To see an example of the difference between expressions with De Bruijn indices and expressions with variable names conversion, first, consider the following expression in CakeML.

```
let
  fun f x = g x
  and g x = f x
in
  f 2
end
```

This expression has two mutually recursive functions and a statement. Evaluation of it would not terminate, and it is obviously not useful, but nevertheless, serves as a simple example. Fig. 4.18 shows the result of calling pat_to_pres_exp on a simplified version of this expression in `patLang`. Before the conversion, the number 0 refers to either the input variable of the functions or to the first function, depending on context. Also, the first function is referred to as either 1 or 0, depending on context. In the result, each of the functions defined is converted to a tuple with three elements: 1) the function's name; 2) the name of its input variable; and 3) the function body. The first function is named "b" and the second "a"[4]. In all three occurrences of App, the two functions are called by name.

```
pat_to_pres_exp 0
 (Letrec t1
   [App t2 Opapp [Var_local t3 2; Var_local t4 0];
    App t5 Opapp [Var_local t6 1; Var_local t7 0]]
   (App t8 Opapp [Var_local t9 0; Lit t10 (IntLit 2)])) =
Letrec t1
 [("b","c",App t2 (Patlang_op Opapp) [Var_local t3 "a"; Var_local t4 "c"]);
  ("a","c",App t5 (Patlang_op Opapp) [Var_local t6 "b"; Var_local t7 "c"])]
 (App t8 (Patlang_op Opapp) [Var_local t9 "b"; Lit t10 (IntLit 2)])
```

Fig. 4.18. Example of replacing De Bruijn indices with variable names. Calling App with Opapp as argument is how standard function application is encoded. Then the first list element is the function, and the second is the argument. In the result, names have been assigned to the mutually recursive functions inside Letrec, and the functions are called by name.

4.3 Web application

The web application lets the user input the source code of a CakeML program, see its representation in the ILs used in the compilation process up to `patLang` and be able to interact with it by tracing expressions between ILs. It consists of a server-side application and a user interface in the form of an interactive web page.

[4]Variable names are assigned in reverse order for mutually recursive functions.

4.3.1 Server-side application

The server-side application consists of a simple server that accepts requests containing source code for CakeML programs. Upon receiving such a request, the web server returns a JSON representation of the IL trees for that program.

The web server is implemented in the programming language PHP, with a simple PHP program that accepts HTTP `POST` requests. The `POST` request should contain a parameter `q` that contains the source code of a CakeML program as a string.

The PHP program stores the source code from the `q` parameter in a temporary file, and then runs a bootstrapped, binary version of the CakeML compiler[5] with the content of that file. The binary compiler produces JSON representations of the IL trees for that program by converting them to `displayLang` and then to JSON. The JSON representations are then returned as the response to the `POST` request. It should be noted that this program was written and is hosted by Magnus Myreen.

4.3.2 Graphical user interface

The web page before any user interaction is depicted in Fig. 4.19. It consists of a text area where the user can input their source code, and a "Compile" button. After a user has input their program and clicked "Compile", the blank area below will contain text representations of each IL, which is illustrated by Fig. 4.20.

The expressions in the text representations are clickable. When the user clicks on an expression in a certain language, and that expression has a trace, that expression becomes highlighted with a light blue background color, as depicted by Fig. 4.21. We will hereafter refer to that expression as the *active expression*, the language which it came from as the *active language*, and the trace of the active expression as the *active trace*. Expressions that are ancestors to the active expression become highlighted with a light red background color, and expressions that are descendants become highlighted with a light green background color. This is illustrated by Fig. 4.22 and Fig. 4.23, respectively. (The texts explaining which language the highlighted expressions are from were added to the figures afterward, and are not present in the GUI.)

When the user clicks on another expression that has a trace, that expression will be the new active expression, and it and its ancestors and descendants will be highlighted accordingly. If an expression without a trace is clicked, however, the currently active expression is deactivated. This causes the highlighted expressions—the active expression, along with its ancestors and descendants—to no longer be highlighted.

4.3.3 Rendering HTML using React components

The GUI of the web application is built using the JavaScript library React, which is described in Section 2.4.

The primary feature of the GUI is the text representations of the IL trees, which are designed to be as close as possible to how they would have been written as

[5]The result of bootstrapping the compiler and creating an executable file from it, that can be run by a computer directly without requiring the HOL interactive environment.

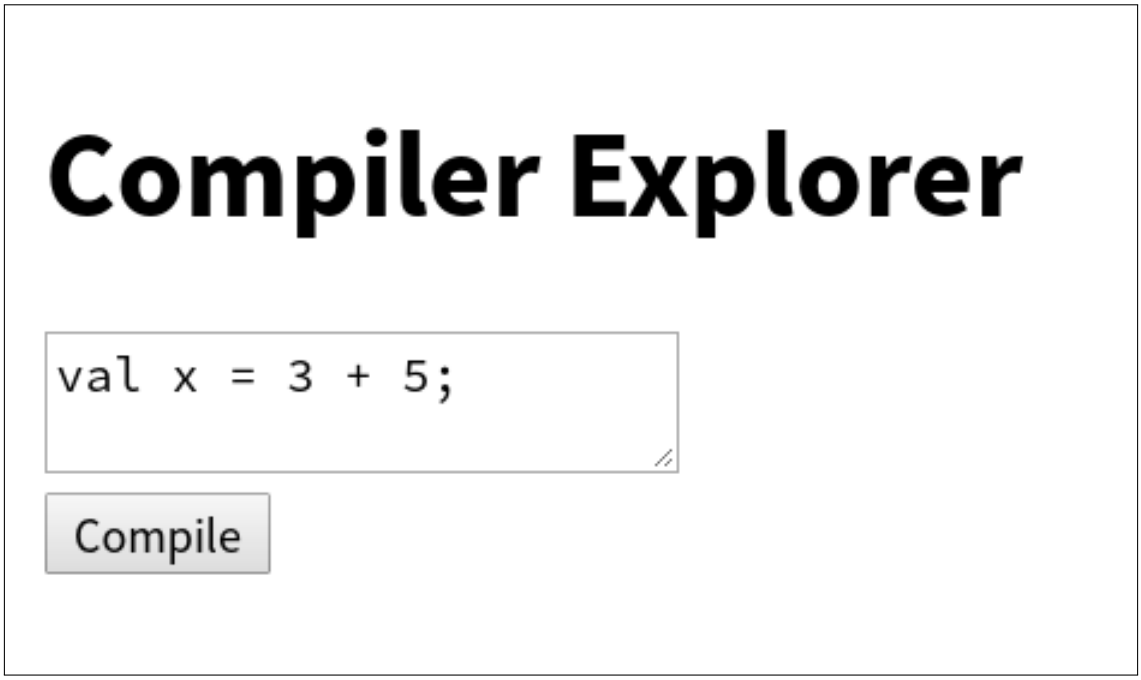

Fig. 4.19. Web application GUI before any user interaction. The example program `val x = 3 + 5;` is pre-filled in the area for source code.

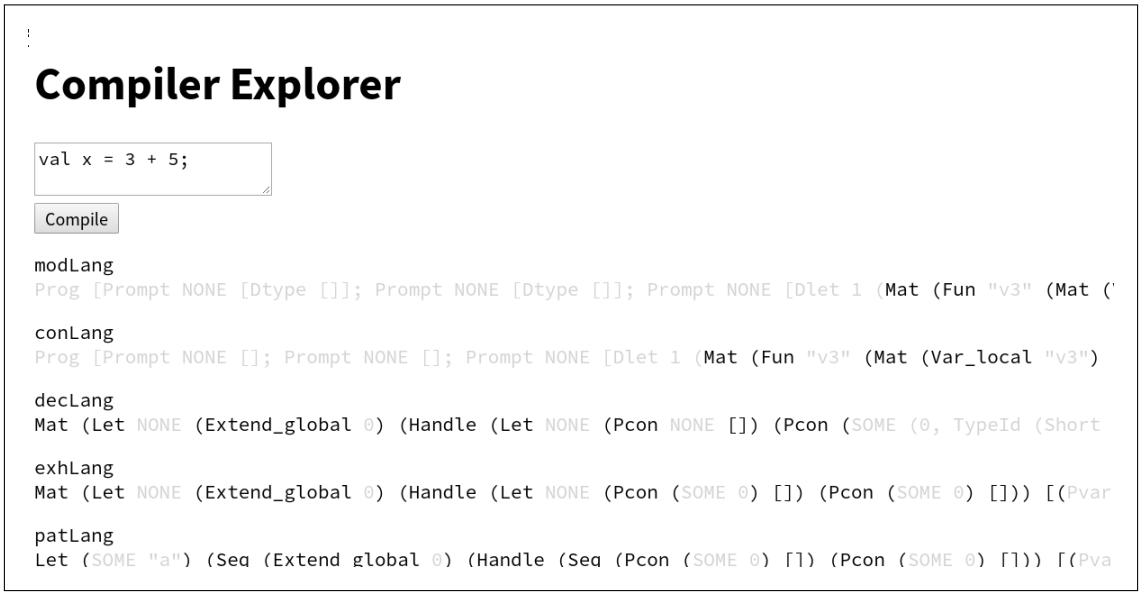

Fig. 4.20. Web application GUI after clicking the "Compile" button. The text representations for the IL trees are visible. Expressions with gray text do not have a trace, while expressions with black text do.

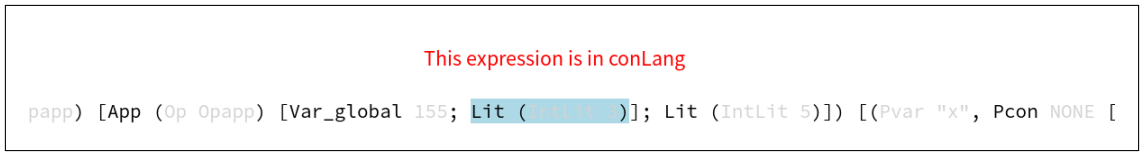

Fig. 4.21. Active expression in the web application GUI. It is highlighted with a light blue background color. The expression is in `conLang`.

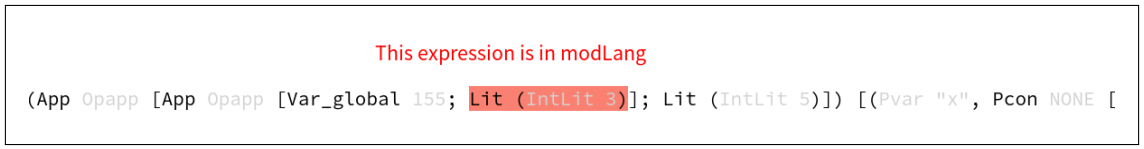

Fig. 4.22. Ancestor expression in the web application GUI. It is highlighted with a light red background color. The expression is in `modLang`.

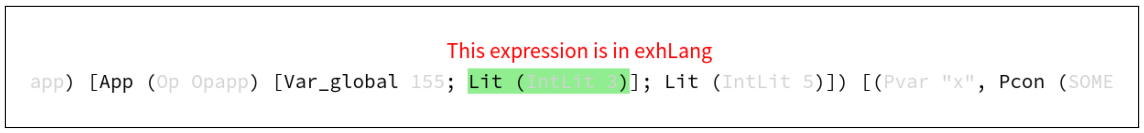

Fig. 4.23. Descendant expression in the web application GUI. It is highlighted with a light green background color. The expression is in `exhLang`.

single expressions in HOL syntax. This means the text representations use the appropriate syntax for constructors, lists, tuples, etc., as well as added parentheses to avoid ambiguity.

All expressions are contained in one HTML `<span>` element each, in a recursive fashion. For example, the expression `Lit (IntLit 4)` would be rendered as `<span>Lit (<span>IntLit 4</span>)</span>`. This does not affect how the expressions are rendered visually [26, Sec. 4.5.28], but it allows for retaining the structure of the IL trees even in the HTML element structure.

Since the IL trees are converted to `displayLang` expressions, and `displayLang` has a well-defined and clear structure, the web application implements rendering of any `displayLang` expression as a tree of HTML elements. This is done with four React components: one top-level component called `<DisplayLangExp>`, and three auxiliary components called `<Item>`, `<Tuple>`, and `<List>`. Together they are able to render any `displayLang` expression as a tree of `<span>` elements.

The purpose of the `<DisplayLangExp>` component is to take a `displayLang` expression, determine whether it is an Item, Tuple, or List, and then delegate the actual rendering to the corresponding `<Item>`, `<Tuple>`, or `<List>` auxiliary component, respectively. The auxiliary components contain logic for rendering their corresponding `displayLang` constructor as text in HOL syntax.

- The `<Item>` component takes two props called `name` and `args`, correspond to a constructor name and its arguments, respectively. It renders each element

in `args` using the `<DisplayLangExp>` component and then returns a `<span>` containing the constructor name and the rendered arguments.

- The `<Tuple>` component takes one prop called `elements`, corresponding to the elements of a tuple. It renders each element in `elements` using the `<DisplayLangExp>` component and then returns a `<span>` containing the rendered elements, separated by commas, and surrounded by parentheses.
- The `<List>` component behaves identically to the `<Tuple>` component but uses square brackets instead of parentheses, and semicolons instead of commas.

4.3.4 Highlighting expressions on click

The expressions that have traces should be clickable to highlight the along with their ancestors and descendants. To do this, each `<span>` containing an expression with a trace has its `onclick` attribute set to a handler which causes that expression to be activated. Additionally, the handler consumes the click event and stops it from propagating to any enclosing `<span>`s by calling the `stopPropagation` method [27, Sec. 3.1].

The expressions in the IL trees, as they are represented as `displayLang` expressions, contain no direct information about which expressions are their ancestors and descendants. This can, however, be calculated by comparing the traces of the expressions, as explained in Section 4.1.3. When an expression is activated, the trace of each expression in the IL trees is compared to the active trace to determine ancestry.

This information is added to the expressions using what we call *tree decoration*. Tree decoration is the process of replacing each node n in a tree with the tuple $(n, f(n))$ for some function f. Each node is thus "decorated" with some extra information that is derived from the node itself. In the web application, tree decoration is performed on the IL trees, so that each expression as "decorated" with a new property called `highlight`, containing information about how that expression is related to the active expression. The value of `highlight` is one of `"ancestor"`, `"descendant"`, `"equal"`, or `"none"`.

At the heart of the web application's implementation of tree decoration is a function called `decorateExp`. It is capable of traversing any `displayLang` expression `exp`, applying a function `func` to any HCO(Item), and storing the result under a new property with name `key`. The web application uses this by using the IL trees as `exp`, a function to determine ancestry as `func`, and `"highlight"` as `key`. The function to determine ancestry compares each expression's trace to the active trace.

The `highlight` property is then used in the `<Item>` component: if the value is `"ancestor"`, the background color is set to a light red color; the value `"equal"` gives a light blue background color; and the value `"descendant"` gives a light green background color.

5
Discussion

In this chapter we will discuss the way we worked, our results, their fit as a solution to the problem we have defined and suggested improvements that can be made to the explorer. We also discuss how we learned the tools used in the development and various difficulties we encountered along the way.

5.1 Planning the project

In the early phases of the project, we had to get a clear picture of what we were to build, learn the compiler tools and then plan our work from there. This section will discuss our experience doing so.

5.1.1 Getting user input

One of the two target groups for the compiler explorer are new developers of the CakeML compiler. However, the compiler explorer never reached a state that allowed for user testing on this target group. This outcome is unfortunate since the purpose of the compiler explorer was, in part, to make it easier for these new developers to learn and understand the compiler. The lack of input from this target group makes it difficult to determine if the envisioned final product would be helpful for new developers.

On the other hand, we did have contact with the other target group of the compiler explorer, namely the current CakeML developers. This target group is small, with the CakeML developer mailing list consisting of 22 persons excluding us, the authors. In particular, we had regular contact with our supervisor who is also one of the CakeML developers. This way we got regular feedback on how well the compiler explorer satisfied the expectations of this target group. Since they are the primary experts on the compiler and have an interest in attracting new developers, we believe that this contact has been enough to ensure that the compiler explorer will be reasonably useful to the new developers.

5.1.2 Learning HOL and CakeML

To start building the compiler explorer, we had to learn both HOL and CakeML. We followed the official tutorial [28]. The HOL4 website [29] states that for beginners the average time it takes to able to use HOL comfortably is about a month[1].

Given this challenge, it is not surprising that we invested much time in learning how to modify the CakeML source code. Getting a working development environment took us several days, and it often broke in ways we were not able to fix ourselves. Sitting down with our supervisor, we continuously learned many tips and tricks that could have saved us much time early on.

The same problem applied to CakeML and the compiler. The source code for the compiler has many specific conventions, lacks comments in many places, and has incomplete documentation, making it hard to enter the project without significant assistance. In the end, we developed an understanding of the core ideas behind the compiler, the purpose of different ILs and compiler passes, but perhaps our knowledge remains somewhat shallow.

Another difficulty was that since the compiler is being continuously developed, the code base has undergone changes during our work. Most notably, line annotations in their current form were not implemented when we started this project, even though a decision had been made to implement them. This meant we had to work without them in the beginning. The result was slower development early on in the project, which set us back in our planning.

5.1.3 Division of labor

Being only three people in the project meant we had little redundancy in personnel. To prepare for the possibility of one team member becoming incapacitated, we decided to try to distribute knowledge fairly evenly. We all learned about HOL and CakeML and worked closely together during the first half of the project.

We grossly underestimated the work required for the web application. Whereas adding traces and outputting JSON for a single language from the compiler was nearly completed on schedule (end of February) the web application never came to the first prototype stage we hoped would be done by the end of the first half of the project. Two group members worked on adding traces to the compiler, whereas one group member worked on the web application, as we had anticipated this to be the easier problem to solve. Even so, we continuously underestimated the time it would take to complete the web application, which meant we did not allocate another team member to work on it until the last month when it was hard to get up to speed quickly enough.

[1] Our supervisor has informed by our supervisor that this estimate likely is for graduate students pursuing a Masters degree or a Ph.D.

5.2 Compiler changes

Regarding the changes made to the compiler, the most significant ones are the introduction of traces and our three ILs. We discuss these, how we handled de Bruijn indices, and the development process, in turn, here.

5.2.1 Implementation of traces

Traces are a simple solution to a complex problem, utilizing only two recursive constructors and one non-recursive constructor to encode position and an additional constructor for turning traces off. The implementation even allows distinguishing different types of origin with little efforts, such as orphan expressions and expressions originating in code.

The structure was chosen specifically to limit the number of constructors, and therefore the number of patterns that need to be handled when working with traces. The downside of the simple structure is that we employ two tricks in our encoding, namely making the source position a series of Cons and encoding orphan expression with a Union. We could, of course, use separate constructors for encoding the start of an expression as either a source position or an "orphan", removing Empty. In the end, we feel that refactoring to using more constructors would be sensible and easy, at least on the compiler side, and it is something we would recommend for future development.

A further consideration that has not been brought up, however, is the performance regarding speed, e.g., for matching traces together in the web application. We decided to not focus on this performance as it is not something that the developers have asked for. Also, traces will not grow arbitrarily large but will be only slightly larger than the number of compiler passes, which rarely increase.

Another issue with our implementation is that it meant we added an extra argument to the constructors in every language we added to explorer. This means multiple changes to the compiler functions, and therefore also to the correctness proofs of the compiler. One option we investigated was to decorate similarly as described in [30], which aims to solve a very similar problem in Haskell. However, the proposed solution makes use of extensible data types, a proposed syntactic feature in Haskell [**?**], [31], and nothing like it is available in HOL.

Traces are designed so that it should be impossible for two expressions to end up with the same trace. However, we have not verified this property formally. This might be dangerous since there is a risk that two expressions which are not related end up having the same trace if our method for building traces is flawed. However, we have seen no such behavior in our testing, and have little reason to believe that such a mistake could have occurred in our work. We also believe we could perform a verification of the uniqueness of traces, but we have not made it a priority.

5.2.2 New intermediate languages

One of our goals was to keep the logic of visualizing the compiler separate from that of compiling programs. As in any software project, we wanted to separate concerns

and not have to do the same thing many times in different places. As a result we introduced our own ILs, `presLang`, `displayLang` and `jsonLang`. These separate the concerns of removing semantics, presenting data and converting to an output format, respectively, and ensure that a single pass to `presLang` is enough to output any IL from the compiler – no further special treatment per language needed.

Also, by introducing `presLang` we did not have to deal with semantics in `presLang`, `displayLang` and `jsonLang` as they can never be run. We realized that by introducing the new ILs, we could compile to our desired output without also introducing new semantics and possibly difficult proofs that would burden the CakeML project as a whole. This choice allowed us to work mostly independently without risk of interfering with the core development team's work.

Further, the introduction of `displayLang` makes sure our solution is extensible with regards to the output format. The design of `displayLang` allows future development to add support for any desired output format with only a few lines of code.

5.2.3 Handling De Bruijn indices

While De Bruijn indices are useful in computing, we believe that displaying bound values as variable names is more intuitive to humans. Therefore, we believe converting De Bruijn indices to variable names was a necessity to make the compiler explorer comprehendible for `patLang`, where values are bound using De Bruijn indices. The conversion results in a presentation which is easier to comprehend because of the usage of ordinary variable names. The downside to the conversion is that it is not a perfect representation of how expressions look in the compiler at that stage. While this might seem counterintuitive to the objective of using the compiler explorer as a tool to learn more about the compiler, it makes for an easier way to follow what is going on. Ideally, in the future, the compiler explorer could switch between different modes where you can choose to either see an IL exactly as is or with modifications for easier understanding. It is worth to be noted that one of the weaknesses of the previous compiler explorer was that the parts with de Bruijn indices were unreadable, see the response of Magnus Myreen in Appendix A.

5.2.4 Testing changes to the compiler

The intention of verified software is to rely on formal proofs rather than traditional software testing—such as unit tests and integration tests—to ensure the correctness of programs. This is the case for the CakeML compiler. However, in the course of this project, we opted to not rewrite proofs to accommodate changes, due to time constraints. Instead, our changes were verified by regularly bootstrapping the compiler and running it.

The proofs we have affected will, however, need to be updated for our changes to be merged into the main CakeML code base. After discussion with the CakeML development team, we have learned that it should be a simple matter to update the proofs since our changes do not affect semantics. It should be sufficient to ignore

the traces in the semantics proofs and to add terminations proofs for functions that evaluate traces.

To informally verify continuously that our code has been working, rather than wait on the slow bootstrapping process, we implemented a small file with actions that can be run manually, and the output inspected for errors. This has been considerably slower than an automated procedure. However, we have not learned the necessary tools to automate such a process with HOL, and we have not considered this testing a large enough priority to allocate time for this learning.

5.3 Web application

The web application we created was a proof-of-concept, only implementing the minimum functionality required. Presently, it is quite slow and difficult to overview. However, the most important technical problems have been solved. In this section, we discuss our experience of building the web application with React and possible performance optimizations.

5.3.1 Using React

We are happy with our choice of using React to create the GUI. The functional programming-oriented approach of React fits particularly well to the type of problem the GUI has to solve, namely rendering a recursive data structure. We, the authors, have previous experience with functional programming, which enabled us to more quickly learn and get familiar with the programming model of React. Had we not had that experience, learning React had been more of a challenge which would likely have delayed the proof-of-concept web application even further.

Additionally, the declarative programming style that React allows for most likely drastically reduced the complexity of the code responsible for highlighting expressions. Since React handles updating the HTML automatically, we could focus our efforts on correctly implementing the logic for calculating ancestry, and not consider exactly how the background color of the `<span>`s should get updated.

5.3.2 Performance

As it stands, the web application takes several seconds from a user click until highlighting is completed. We find it to be detrimental to the user experience. However, there are ways we could improve performance by adapting the compiler output and the rendering process, by utilizing the fact that the compiler runs faster than the web application, and by using smarter ancestry calculation algorithms.

An improvement that could be made on the compiler side is to remove traces before they are output and replace them with a structure which is faster to traverse. In the pass from `displayLang` to `jsonLang`, every trace could be converted to a single, unique number. Then instead of a trace, each expression would contain said number, and two lists: one for the numbers which are its ancestors, and one for those which are its descendants. Also, the compiler would output a table of which source code positions correspond to which trace number. This would mean that comparing

traces would become a problem of comparing numbers, which is an improvement from $O(n)$ to $O(\log m)$, where n is the size of the largest trace, and m is the total number of unique traces. Since there are, in the worst case, as many levels to a trace as there are compiler passes, and there are nearly 30 compiler passes, this should give some improvement.

On the web client side, a further optimization would be to create a lookup table which associates each unique number with all the expressions which have that number as its trace. As the page is rendered and elements are created, a pointer to each element with a trace number could be added to a list of such pointers in the lookup table. This would mean that instead of traversing the entirety of the IL trees every time a new expression is clicked, highlighting would simply be a matter of looking at the ancestor and descendant numbers and looking up the corresponding elements, highlighting them, without traversing unrelated elements at all.

Another optimization that could be done on the web client side is to be more cautious with which parts of the IL trees that are rendered by default. At present, the web client unconditionally renders the entire IL trees, including the entire prelude which encompasses a vast portion of the entire IL tree. As we discussed in Section 3.2.3, we have discussed several methods for marking the prelude, which could be used to not render it by default but rather on demand. This would most likely speed up both the initial render (when the compilation finishes) and subsequent renders (when the user clicks on expressions) due to the browser not having to render a large amount of HTML elements.

5.4 Suitability for intended uses

As the survey responses in Appendix A indicate, especially the answers to question 3, the development team want to be able to visualize where code goes and explain to newcomers how the compiler works. Magnus Myreen also says that it was hard to read intermediate code with De Bruijn indices and that it was hard to find the relevant parts of a program. Scott Owens further hopes that he can get intermediate output for a document explaining the compiler which explains the compiler.

Currently, the web application has issues that would not make it fully suitable for this intended use yet. Firstly, it is hard to overview the output as the user must scroll horizontally to find highlighted code, and highlighted sections of the code are not lined up. Secondly, the output lacks pretty-printing, making it hard to parse for a human reader. Thirdly, it does not cover the whole compiler from source to machine code, but only the initial five ILs. Fourthly, prelude code always gets printed, making the output large even for small programs. Lastly, highlighting is slow. Each of these is a problem that we know how to address, but which was left unsolved due to time constraints. All of these problems are fairly easy to solve, and we outline the solution to all of them in this chapter. We have little doubt they will be solved as the development of the compiler progresses after the end of this thesis project.

Other than supporting developers, the motivation for building the compiler explorer was to give insight into how the CakeML compiler behaves. While we believe the completed compiler explorer will be a great tool to this end, one important

drawback of our solution is the changes we made to the compiler. Our addition of traces to constructors means that every expression has an extra argument, the purpose of which may not be clear to a newcomer to the CakeML project, and which is itself not shown in the compiler explorer. We believe this initial confusion to be modest and passing, as the purpose is easy to explain and the format is consistent through all modified languages, but it is nevertheless a drawback to our solution.

5.5 Effects on society as a whole

The CakeML project is an important research project. While introducing the CompCert compiler, which is also verified, Leroy [2] states that if in the future, formal methods are common for checking source code, "the compiler could appear as the weak link in the chain".

Seeing how software errors could have fatal consequences when occurring in an autonomous car or medical equipment, verified compilation might very well take a central role in the development of safety-critical software. The future of this field may in large determine whether we trust – or should trust – the software we build in the future and increasingly trust with our lives. In the end, a verified compiler may prevent disasters.

As it stands, however, there are few verified compilers, CompCert and CakeML being two prolific examples. These are both developed by small teams. With a compiler explorer at hand, it will be possible to more easily expand the CakeML team with newcomers, increasing the amount of work put into CakeML, and improving its prospects for the future.

By helping the core developers find optimizations the chances of adoption of the CakeML compiler increases, as performance of software is often an important consideration. By helping the developer team identify optimizations, it is our hope that they can prove that verified compilers can realistically be good at optimizing, furthering the field of verified compilers as mainstream adoption becomes easier.

At the very least, of course, the compiler explorer can make the work of the developers easier, and perhaps inspire similar projects for other products, making the work of understanding other compilers less cognitively burdensome for those working on them.

5.6 Future work

In this section, we will discuss different improvements and new functionality that could be added which we believe would improve the compiler explorer. We discuss a different improvement in each section, and order the sections according to our perceived importance, from most important to least important.

5.6.1 Improving overview

The current user interface prints each AST horizontally and does not line up related expression. This means that when the user clicks an expression, highlights might

appear off-screen and thus be tricky to find. To cope with this, our suggested solution is to display each AST with independent scrolling. Also, we suggest that the web interface should automatically scroll to expressions when they become highlighted and have the possibility to step through all highlighted expressions.

5.6.2 Source code highlighting

Currently, the web application does not provide any interaction with the input source code. The developer team, through our supervisor, have expressed a desire to be able to see where in the source code an expression in an IL has originated. Indeed, this is a reason that we used the line annotations from the source code in our trace.

Allowing the user to click on expressions in the source code and have corresponding expressions in ILs highlighted, and vice versa is a matter of improving the web application. As it stands, the necessary information to implement this functionality in the web application is provided in the output from the compiler.

Highlighting the source code when the user clicks an IL would be a simple matter. By looking at the clicked expression's trace, we could find all sequences of four Cons terminating with Empty. These would all indicate start and end positions of the expressions to be highlighted, as explained in Section 4.1.2. For example, if an expression with the trace shown in Fig. 4.6 was clicked, the characters from row 1, column 2 up to and including row 3 and column 4 would be highlighted in the source code, as would the characters from row 7, column 8 to row 9 column 10.

In the reverse situation, when the user clicks on an expression in the source code, the procedure would be different. Firstly, since clicking on the source code means clicking on a specific character in a string of text, the row and column of the clicked character would have to be noted by the web application. Secondly, the web application would have to search through the traces in `modLang` to find the smallest trace that includes the clicked character. In this context, *smallest* refers to the number of characters included in the portion of the source code that is encoded by the last four Cons's in a trace. The smallest trace would then be set as the activated trace, and highlighting would be performed as usual.

5.6.3 Pretty-printing

The text representations of the IL trees in the web application are very hard to read, as they simply stretch from left to right without line breaks. An improvement to readability would be to do pretty-printing that would mimic what a programmer would write.

The most naïve way to do pretty-printing would be to print an IL tree as a regular tree by doing the following.

- After each constructor name, or opening bracket or parenthesis, do a line break and increase the indentation level.
- After each argument to a constructor, or element in a list or tuple, do a line break and keep indentation level.

- After completing an argument list or closing a bracket or parenthesis, do a line break and decrease the indentation level.

Below is an example of what the result would look like. Consider the following code.

```
Foo (Bar 2) (t1, t2) [e1; e2; e3]
```

By the steps above, the above code would be pretty-printed as follows.

```
Foo
    (Bar
        2)
    (
        t1,
        t2
    )
    [
        e1;
        e2;
        e3
    ]
```

An even better way to pretty-print the code would be to consider line length as well. For example, a list that would fit on a line should get printed without line breaks, and otherwise receive line breaks only as needed. The same would go for constructors, but these might get special treatment depending on the constructor so that for example a Let would be printed in a way that mimics how `let` is often written in code.

Making this change would also mean that the layout of the web application should change, from displaying each IL on its own line to showing them in columns. As each IL would then occupy a lot of space both vertically and horizontally, this would be a good time to switch to only displaying two or three ILs at a time.

There exist libraries for general pretty-printing of text, such as John Hughes' `pretty` library for Haskell [32]. However, we have not found any suitable library of this type for JavaScript. Furthermore, if such a library exists, it might prove difficult to be integrated with React and the rendering of HTML elements that have `onclick` functionality and similar features.

5.6.4 Optimizations

The performance considerations mentioned in Section 5.3.2 should be addressed by making the suggested optimizations. Currently, rendering takes quite some time, which is detrimental to the user experience.

5.6.5 Tracing the entire compiler

At the moment the compiler explorer traces and displays all ILs from `modLang` up to and including `patLang`. This is in itself valuable, but developers that work on lower level ILs will not be able to benefit fully from the explorer until we support all compiler passes. However, this is a straightforward process and one which is already in progress.

Adding traces to a new IL is sometimes a complex matter, but has so far has proven feasible, and a project that does not require more than a day's work. Complications arise when transformations are not straightforward, e.g., call many helper functions to which the trace then needs to get passed, as the resulting expressions may be nested and hard to annotate after their creation. If the traces are added by a developer who is familiar with the IL in question, and thus has a good overview of the compiler passes performed there, we imagine it would often be quicker.

Extending `presLang` to accommodate a new IL is not difficult, but can be time-consuming, as new constructors need to be added and modified. For example, `closLang`, the next IL after `patLang`, has the constructor Tick which does not occur in any earlier IL, and its App constructor looks different from all previous ILs. These issues have occurred in our implementation process, and we have solved and uniformly documented them, so this too should not take more than a day's work for an IL. However, maintaining `presLang` when the compiler changes might become burdensome. It might be feasible to auto-generate `presLang` and conversion to and from it, but we have yet to find a method for this generation which can handle even a single IL and which would not itself be as difficult to maintain as the current code.

The web application is agnostic as to which languages it gets. Thus, if the compiler explorer outputs JSON for a new IL, the web application will be able to accept and display it without modification.

5.6.6 Refactoring `tra`

As discussed in Section 5.2.1, we believe traces could be refactored to have more expressive constructors, such as creating one constructor for source code position, and one for orphans. Orphans might be removed if the source code position for declarations is used instead, as is now possible. If tracing prelude code is implemented, a special constructor for prelude expressions could be added.

5.6.7 Tracing prelude code

As mentioned in Section 3.2.3, the compiler adds a prelude to every program it compiles, which is fairly large and has no traces. It was only late into this project that we started using a web server with a bootstrapped compiler, which was the first time this problem was encountered. We have since then discussed possible measures to handle the prelude code, as well as the display of different ILs in general, but time has not permitted us to implement a solution. However, we consider this the next important step in the explorer project.

Our proposed solution to this problem is to create a new, special trace constructor for prelude traces, given that the refactoring proposed in Section 5.2.1 has

been performed. If not, a special prelude trace could be created, one which would never match an existing trace, much like the orphan traces introduced in Section 4.1.4. Using this new trace as base, we could traverse the prelude, assigning unique numbers to each expression by wrapping the base in Cons.

In the web application, we could then identify expressions with these special traces and fold them together, so that the user may view them, but not see them by default.

6
Conclusion

The goal of this project was to develop a tool for exploring how the CakeML compiler transforms its input code, compiler pass for compiler pass. The motivations for this were twofold. Firstly, current developers of CakeML should be able to more easily identify possible optimizations the compiler could make. Secondly, newcomers to the compiler should be able to get a good understanding of how the compiler works quickly.

The final product, called the compiler explorer, is a prototype tool for visualizing the compiler transformations. It consists of a web application in which the user can see the state of a compiled program after each compiler pass. By clicking on expressions in the program, corresponding expressions after other compiler passes get highlighted, and the user can then see how the compiler has applied transformations to specific expressions.

With our project, we have solved the general problems of building a complete compiler explorer. Extending our solution to cover the entirety of the compiler will a straight-forward matter.

To solve the issue of relating expressions with each other after different passes, we invented the concept of traces, which is a small datatype which can be used to encode all the necessary information in a simple tree structure.

We also needed to output the necessary information from the compiler in a format that the web application could consume. To this end, we introduced three new intermediate languages. These are semantics-free and thus do not need to be verified. In this way, we interfere minimally with the development of the compiler in general and add very little overhead regarding proofs.

To the best of our knowledge, this is the first product which shows the internal transformations of a compiler with this level of interaction, where the user is shown not only the state after each transformation, but also how these transformations happen. By connecting expressions in different intermediate languages together in the web application, we expect it will be much easier to get a firm grasp of the CakeML compiler's inner workings.

During this project, we have learned much about compilers and their inner workings. We have also gained insight into the formal methods required to verify a compiler and the research contributions that the CakeML project makes. Even though learning the necessary tools for this project is known to be hard, we have been able to make great progress thanks to the help and mentoring of our supervisor and the CakeML development team.

Bibliography

[1] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel, "Compositional CompCert," *POPL: Principles of Programming Languages*, vol. 50, no. 1, pp. 275–287, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2775051.2676985

[2] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009. [Online]. Available: http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf

[3] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A verified implementation of ML," in *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2014, pp. 179–191.

[4] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "A new verified compiler backend for CakeML," in *ICFP '16: Proceedings of the 21th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, Sep. 2016, pp. 60–73.

[5] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," *SIGPLAN Not.*, vol. 46, no. 6, pp. 283–294, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/1993316.1993532

[6] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan, "Functional big-step semantics," in *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016*, ser. Lecture Notes in Computer Science, P. Thiemann, Ed., vol. 9632. Springer, 2016, pp. 589–615.

[7] R. Kumar, R. Arthan, M. O. Myreen, and Owens, "Self-Formalisation of Higher-Order Logic: Semantics, Soundness, and a Verified Implementation," *Journal of Automated Reasoning*, vol. 56, no. 3, pp. 221–259, 2016.

[8] "CakeML: A Verified Implementation of ML," https://github.com/CakeML/cakeml, Accessed: May 2, 2017.

[9] M. Myréen, 2017, [Electronic image]. To be published. Provided by Magnus Myreen.

[10] N. G. De Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem," vol. 75, no. 5, pp. 381–392, 1972.

[11] K. Chaudhuri, "Illustration for De Bruijn index," 2009, Accessed: April 4, 2017. [Online]. Available: https://commons.wikimedia.org/wiki/File:De_Bruijn_index_illustration_1.svg

[12] M. Norrish and K. Slind, "The HOL System Description," 2011. [Online]. Available: https://sourceforge.net/projects/hol/files/hol/kananaskis-11/kananaskis-11-description.pdf/download

[13] "React - A JavaScript library for building user interfaces," https://facebook.github.io/react/, Accessed: May 9, 2017.

[14] "Rendering Elements," https://facebook.github.io/react/docs/rendering-elements.html, Accessed: May 12, 2017.

[15] "Components and Props," https://facebook.github.io/react/docs/components-and-props.html, Accessed: May 12, 2017.

[16] "CakeML," https://cakeml.org/, Accessed: May 4, 2017.

[17] D. Sarkar, O. Waddell, and R. K. Dybvig, "A Nanopass framework for compiler education," *Journal of Functional Programming*, vol. 15, no. 05, p. 653, 2005.

[18] "LLVM Visualization Tool User Guide," https://llvm.org/svn/llvm-project/television/trunk/docs/UserGuide.html, Accessed: April 28, 2017.

[19] "Compiler Explorer," https://github.com/mattgodbolt/compiler-explorer, Accessed: May 2, 2017.

[20] "CakeML Compiler Explorer," https://cakeml.org/explorer.cgi, Accessed: April 4, 2017.

[21] "CakeML: A Verified Implementation of ML," https://github.com/Saser/cakeml, Accessed: May 11, 2017.

[22] "Saser/compiler-explorer-react," https://github.com/CakeML/cakeml, Accessed: May 11, 2017.

[23] J. McCarthy, "Recursive functions symbolic expressions and their computation by machine, Part I," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[24] T. Bray, "The javascript object notation (JSON) data interchange format," 2014. [Online]. Available: https://tools.ietf.org/html/rfc7159.html

[25] "cakeml-github-reference," https://github.com/CakeML/cakeml/blob/master/documentation/reference.tex, Accessed: May 11, 2017.

[26] W3C, "HTML5 – A vocabulary and associated APIs for HTML and XHTML," https://www.w3.org/TR/html5/single-page.html, Accessed: May 10, 2017.

[27] ——, "UI Events," https://www.w3.org/TR/uievents, Accessed: May 10, 2017.

[28] M. Norrish and K. Slind, "The HOL System TUTORIAL," 2014. [Online]. Available: https://sourceforge.net/projects/hol/files/hol/kananaskis-11/kananaskis-11-tutorial.pdf/download

[29] "HOL Interactive Theorem Prover," https://hol-theorem-prover.org/#doc, Accessed: May 9, 2017.

[30] S. Najd and S. P. Jones, "Trees That Grow," pp. 1–20, 2016. [Online]. Available: http://arxiv.org/abs/1610.04799

[31] "Exstensible datatypes - HaskellWiki," https://wiki.haskell.org/Extensible_datatypes, Accessed: May 9, 2017.

[32] "Pretty : A Haskell Pretty-printer Library," https://github.com/haskell/pretty, Accessed: May 31, 2017.

A

Survey responses

In May 2017, we conducted an informal survey to verify that the initial requirements we had set up were valid. What follows is the email we posted to the developer mailing list, followed by the response we got, and from who. The responses have been cut down to only the answers to the posed questions.

```
Hi devs,

If you have time, we'd like to hear some opinions from you. We'd
   really appreciate it if you had time.

We're writing our thesis about the new compiler explorer, which
   allows the tracing of expressions, so that you can
a) get the program you are compiling printed at each intermediate
   language and
b) click expressions in one intermediate language and have the
   corresponding ones highlighted in another language.

So we have three questions to you. A sentence or two one each
   would be enough.

1. Did you use the old compiler explorer?
   (https://cakeml.org/explorer.cgi)

2. If yes, then for what? If no, why not?

3. What do you imagine you could use the new explorer for that you
   couldn't use the old one for?


We'd like to quote you on your answers, if that's alright.

Cheers,

Rikard
```

Response: Ramana Kumar

```
    1. Did you use the old compiler explorer?
       (https://cakeml.org/explorer.cgi)

Yes.
```

2. If yes, then for what? If no, why not?

To see if it worked. To see what intermediate language programs looked like. To debug or better understand compiler passes while trying to verify them.

3. What do you imagine you could use the new explorer for that you couldn't use the old one for?

Improving the performance or size of generated code, by being able to visualise and improve one's understanding of intermediate outputs, including while tweaking compiler passes. The old explorer could be used for that, but it's more relevant in the new one because the new one supports the version 2 compiler which has more opportunities for optimisation.

Also, to demonstrate the CakeML compiler's behaviour to newcomers. The old one could be used for that too. But the old one was slow - required running compilation in the logic. I hope the new one can also work with the bootstrapped compiler outside the logic so can run faster.

Response: Michael Norrish

1. I tried using the old explorer once or twice
2. I used it to try to understand the semantics of the various intermediate languages
3. I expect to be able to better understand how expressions at one level correspond to code at later levels.

Response: Magnus Myreen

1. Did you use the old compiler explorer?
(https://cakeml.org/explorer.cgi)
Not really, only a few times.

2. If yes, then for what? If no, why not?
The most significant reasons were that the de Brujin indices were unreadable and it was hard to find the relevant parts of the displayed output.

3. What do you imagine you could use the new explorer for that you couldn't use the old one for?
I hope to be able to use it to look at the intermediate forms in order to spot patterns in the code that could be improved. It is important to understand where the suboptimal intermediate code comes from and where it goes.

Response: Yong Kiam

```
 1. Did you use the old compiler explorer?
    (https://cakeml.org/explorer.cgi)

Yes, I wrote it.

2. If yes, then for what? If no, why not?

Initially, we noticed 1-2 performance bugs in the compiler with it.

Nowadays, I mainly use it for astPP, i.e.\ to pretty print AST out
   to concrete syntax

3. What do you imagine you could use the new explorer for that you
   couldn't use the old one for?

The old explorer hasn't been updated for the new CakeML compiler.
   I've rewritten custom pretty printing for parts of the lower
   level languages multiple times when debugging/tracing through
   them.

This new explorer should help a lot with that.

The operation you describe (click an expression and see how it
   compiles) sounds useful too. (I also do that by hand when
   tracing stuff)
```

Response: Scott Owens

```
> 1. Did you use the old compiler explorer?
   (https://cakeml.org/explorer.cgi)

I tried it out a few times, to see if good BVL code was being
   generated for specific programming idioms. I never really got
   it to work for a few reasons. The biggest were the output being
   hard to read and the whole thing silently failing on parse
   errors.

> 2. If yes, then for what? If no, why not?

See above.

> 3. What do you imagine you could use the new explorer for that
   you couldn't use the old one for?

I hope that I can use it to generate examples at all of the ILs to
   incorporate them into a document explaining what the compiler
   does.
```