Yet another SPL compiler

Orpheas van Rooij Joshua Steinmann

August 5, 2022

Contents

Chapter 1

Introduction

1.1 Language choice

Our language choice for implementing the SPL compiler is Haskell. The reasoning behind this choice can be boiled down to three aspects of Haskell.

Firstly, it is a mature language backed by an industry level compiler, GHC. The compiler has existed for decades and as a result it includes all the necessary productivity features such as language server support, helpful error messages and a rich library ecosystem. Additionally, GHC provides a plethora of language extensions such as rank 2 functions, multi-parameter type classes, GADTs, and even preliminary support for dependent types, extensions that might come in handy when developing a compiler.

Secondly, it is a functional programming language which supports favorable *functional* features such higher-order functions, data immutability, currying, adhoc and parametric polymorphism, algebraic data types, pattern matching and many more. These features make the programming effort much more enjoyable and is frankly the primary reason behind our language choice.

Lastly, it is a statically typed language with type inference which gives us type safety guarantees and increased productivity. For instance, we can ensure our program does not suffer from type errors and we can rapidly reformat code whilst the type checker simultaneously spots the places where the reformatting is inconsistent.

1.2 Simple Programming Language (SPL)

This section explains the basic concepts of our version of the SPL Language – which we name MySPL – and gives some code examples with explanations. In essence, MySPL can be described as an imperative language with higher-order function support, polymorphism, and overloading for a few built-in operators.

1.2.1 Types

MySPL has the basic types void, boolean, integer and character, and also supports type variables, tuple, list and function types.

```
Void    Bool    Int    Char    a    (type, type)    [type]    type* -> type
```

The Tuple, List and Function type constructor take types themselves to form a new type. These composite types can be of any arbitrary depth. Hence a tuple of an integer and a boolean is considered a type, so it is possible to have a list tuples or a polymorphic function that returns another function.

```
[(Int, Bool)]    a b -> (a -> b)
```

1.2.2 Variable Declarations

Types represent the domain of a value stored in a certain placeholder. Variables represent these placeholders. Thus variables are annotated with types to restrict the range of values that can be stored. Variables are declared by giving the type, the name and an initial value.

```
Char c = 'c';
Int i = 5;
Bool b = True;
(Int, Bool) t = (3, False);
[Bool] bs = [];
(a -> Void) printCopy = print;
```

It is also possible to not specify the type in the code, and let the compiler automatically infer the type for you. Instead of the type name the keyword *var* is used then.

```
var v1 = [];
var v2 = 5;
```

Variable declarations can be declared at the top level of the file (global variables) or anywhere inside a function body (local variables). Standard rules of variable shadowing apply – local variables shadow function arguments which in turn shadow global variables with the same name.

1.2.3 Basic Expressions

To make a program useful it needs to be able to compute values from other values. MySPL is able to do so by composing expressions and evaluating them. Below, we describe the allowed expressions in MySPL. Note that these expressions can be composed to form larger ones as long as the types match.

Integer Expressions

Integer expressions represent computations with integer numbers. They can be a literal integer, its negation, or an addition, subtraction, multiplication, division, exponentiation, and modulo operation.

```
3   -3    3 + 4    7 - 5    3 * 4    2 / 4    2 ^ 3     10 % 3
```

Logical Boolean Expressions

Another class are the boolean operators "negation", "logical and" and "logical or".

```
!False    True && True    False || True
```

Note that logical boolean connectives are evaluated lazily.

Field Selector Expressions

Special syntax is provided to easily access the head (hd) or the tail (tl) of a list, or the first (fst) or second (snd) part of a tuple. These are called "field selectors" and can be composed to access a deeper part of the structure.

```
[1].hd    a.tl    (1,'a').fst   b.snd   c.hd.fst.snd    (foo()).snd
```

List Operator

Lists can be constructed in three ways, using the special list syntax, using the string syntax (for list of characters) and lastly using the cons operator. The cons operator (:) allows to prepend the left argument to the right argument which must be a list. For instance the three expressions below represent the same expression:

```
['h', 'e', 'l', 'l', 'o']
"hello"
'H' : 'e' : 'l' : 'l' : 'o' : []
```

Comparison Operator

Comparison operators take two arguments of the same type and produce a boolean value. The equality comparison works for all types.

```
3 == 4    True != False   [1,2,3] != [1,2]
```

Equality on lists works by comparing each corresponding elements in the two lists and checking whether all elements are equal. If one of the two lists is shorter then the lists are treated as not equal.

Comparisons like "less", "greater", "less or equal" and "greater or equal" also work for all types.

```
3 < 4    5 > 3    False <= True    (1,2) >= (0,2)    [1,2,3] <= [3,4,5]
```

Just like the equality on lists and tuples, comparison for composite types work in a positional way. For instance, the expression $[1, 2, 3] <= [3, 4, 5]$ yields true since the two lists have the same length and each pair of corresponding elements satisfies the inequality constraint. On the other hand, the expression $[1, 2] <= [3, 1]$ yields false since $2 <= 1$ is false. In essence, comparison for lists and tuples is not done in a lexographical way but in a positional way.

Lastly, booleans can also be compared. Here the ordering is that False is less than True.

Nested Expressions

Expressions can also be combined and nested. The operators have the following order of precedence.

$||$ $\prec$ $\&\&$ $\prec$ $==, !=$ $\prec$ $<, >, <=, >=$ $\prec$ $:$ $\prec$ $+, -,$ $\prec$ $*, /, \%, \hat{}$ $\prec$ $!$ $\prec$ $-$

The operator with the highest precedence is evaluated first. In case of equal precedence expressions are evaluated from left to right. The only exception is the application of parenthesis.

```
3 + 4 * 7 / (3 + -4)  : []
```

is the same as

```
( 3 + ( (4 * 7) / (3 + (-4)) ) ) : []
```

1.2.4 Assignments

The values resulting from expressions can be captured and reused by making assignments to variables. Those look similar to variable declarations, but they do not specify the type of the variable as they use already declared variables. The variable on the left gets the value computed on the right. It is also possible to assign to a specific field (using field selectors) of the variable such as to the head of the list or to the second part of a tuple.

```
i = 3 + 4;
b = True || False;
c = 'c';
l.hd = 'a'
t.snd.hd = 3;
```

It is important to note that the type of the result of an expression must match the type of the variable it is assigned to. Otherwise a type error is produced.

```
Int i = 0;
i = 'c';
i = 5;
```

Here the first assignment after the declaration is not valid as i has type integer and is assigned a value of type character. The second assignment is fine however.

1.2.5 Functions

MySPL supports higher-order functions, thus functions are treated as first-class citizens. They can be partially applied, can be an argument or the result of a function and can be stored in variables.

1.2.6 Function declaration

A function is declared by giving it a name, a list of argument names, a type and a function body. In the function body the return statement ends the function execution and returns the value to the right of the return statement or no value at all (return type `Void`).

```
sum(a, b) :: Int Int -> Int {
    Int result = 0;
    result = a + b;
    return result;
}
```

The type can also be omitted to let it be inferred by the compiler.

```
subtract(a, b) {
    Int result = 0;
    result = a - b;
    return result;
}
```

In case a function does not return a value there is the special return type Void.

```
nop() :: -> Void {
    return;
}
```

Rank one polymorphic functions can also be defined. Type variables can be used to indicate in the function type that a function works for all types. Alternatively, the function type can be skipped altogether and the compiler will find the most general polymorphic type for it.

```
map(f, xs) :: (a -> b) [a] -> [b] {
    if (isEmpty(xs))
        return [];
    else
        return f (xs.hd) : map(f, xs.tl);
}
```

1.2.7 Function calls

A function can be called, by stating the name and giving the right number of arguments with the correct types.

```
min(3,4)
```

Alternatively, it can be partially applied by giving fewer arguments than the function needs to be evaluated.

```
(Int -> Int) minOf3 = min(3)
```

Partially applying a function with no arguments is not allowed though. For instance, the expression `min()` is not allowed as it represents a call to a function that takes no arguments.

Additionally, the left-hand side of a function call does not have to be a function identifier. It can be another expression too as long as the expression evaluates to a function. This expression must be wrapped in brackets for it to parse.

```
(foo())(3,4)
```

1.2.8 Control flow statements

MySPL supports the conditional control flow statements $if - else$ and *while* loops.

```
if (cond) {some statements}
if (cond) {some statements} else {some statements}
if (cond) statement else statement
while (cond) statement
while (cond) {some statements}
```

Each statement in the body can be a function call, an assignment or a another control flow statement. Statements need to be terminated with a semicolon apart from *if* and *while* statements. Additionally single-statement *if* and *while* statements can be declared without enclosing curly braces.

In the case of ambiguity resulting from a nested single-statement *if* (dangling else), inner-most statements have higher precedence than outer-most statements. For instance the following two statements are equal.

```
if (x)
    print(3);
else
if (y)
    print(4);
else
    print(5);
```

```
if (x) {
     print(3);
} else {
     if (y) {
         print(4);
     } else {
         print(5);
     }
 }
```

Chapter 2

Lexing & Parsing

2.1 Lexing

This section describes and motivates the implementation of the lexer we use.

2.1.1 Purpose and function

The lexer takes SPL source code and produces a list of SPL tokens. To improve the error handling in the latter stages of the process, each SPL token also contains location information such as the line and column number by which it appears in the source code together with its corresponding ending location.

Single line and multi-line nested comments are ignored at this stage. Additionally, invalid symbols or variable/function names, that do not adhere to the naming conventions set by the SPL grammar, produce errors and stop the lexing process. Helpful error messages are produced that indicate which token exactly (together with location in source code) produced the fatal error.

The lexing step allows the parser to work with higher level tokens instead of individual characters and words which makes error handling at the side of the parser easier and more readable while at the same time simplifies the parser.

2.1.2 SPL Tokens

This section presents the set of SPL tokens that is produced by the lexer

Keyword Tokens

The first group of tokens are the keywords specified by the SPL syntax. Tokens in this group concern the program control flow.

- var
- if
- else
- while
- return

Type Tokens

The second group of tokens are the basic type names used in SPL.

- Char (simple, single character)
- Bool (truth values true and false)
- Int (integers)
- Void (used for showing that a function does not return a value)

Symbol Tokens

The third group of tokens are the symbols which have a defined functionality. Symbols that are not listed below cannot be converted to a symbol token.

$$Symbols = \ ! \ : \ | \ . \ , \ \& \ = > < \ \% \ * \ / \ \hat{} \ - \ + \ \{ \ \} \ [\ ]$$

Value Tokens

The fourth group of tokens contains all the literal values for the basic types. For booleans these are $True$ and $False$. Characters are just single characters (including non-printable characters) and integers can be one or more digits.

String Tokens

The fifth group of tokens are the string tokens. These tokens start and end with a double quote and represent strings in the SPL language. We note that non-printable characters inside a string are parsed as normal characters, thus the newline string "$\backslash n$" represents the two character string of $'\backslash'$ and $'n'$.

Identifier Tokens

The last group of tokens are the identifiers used to name variables and functions. Identifiers consist of a character followed by any amount of characters, digits or underscores.

2.1.3 Lexer Generator

Implementation-wise we use the lexer generator *Alex* which is written in Haskell. Alex allows us to specify the different types of tokens by simply using regular expressions, and handles the hard-work of string processing by itself. Additionally, different interfaces of the generated lexer are exposed to the user which allows for extensible customization.

By utilizing Alex's "monadUserState-bytestring" wrapper, we can seemingly utilize bytestrings to handle the tokenization instead of regular strings which can be very performance costly. Additionally, the exposed "monadUserState" wrapper generates a lexer that works in a State-like monad way which can be easily customized. We have extended the state that gets passed around to include the source file name and the source file contents. Additionally, we have modified the way error messages are generated by providing a custom function that utilizes these extra state information to create meaningful error messages.

Alex's "monadUserState" wrapper also enables us to easily support nested multi-line comments by tracking the comment depth inside the state. For instance, the following nested comment is valid:

```
/* outer comment begins
    /* this is a comment inside another comment */
   outer comment follows
*/
```

2.2 Parsing

This section describes and motivates the implementation of the parser. In general, our parser is a top-down LL(1) parser that has been implemented using parser combinators. To handle left recursion we have rewritten the grammar by left factoring it. Similarly operator associativity and precedence rules have been incorporated to the grammar.

2.2.1 Purpose and function

The parser turns the list of tokens produced by the lexer into an abstract syntax tree that represents and respects all the fixity and associativity rules given by the SPL syntax. The AST represents the source code in a structured format that removes any ambiguity (deduced from the concrete parser rules) and contains only the necessary information in a minimal format.

2.2.2 Precedence and Associativity

To handle different precedence of operators in the SPL language we have rewritten the grammar to take into consideration the precedence rules. Appendix A shows the resulting grammar which takes into consideration the precedence rules. For instance, since the logical OR operator (`||`) has lower precedence that the logical And operator (`&&`), we firstly parse the expression `LAndExpr` once followed by a greedy parse of the expression `( '||' LAndExpr)*`. This solves the precedence problem and additionally the greedy parse makes sure that we parse as much as possible.

To handle left, right and non-associativity of operators, we use the standard `pChainR`, `pChainL` and a custom `pNonAssocOp` higher-order parsers. The former two parsers take two parsers – a parser *p* and a fusion parser *op* – and use these to greedily parse sentences of the form `p ( op p )*`. Depending if the `pChainR` or `pChainL` is used, the parse results will be composed as `(p op (p op p)` or as `(p op p) op p`.

On the other hand, the `pNonAssocOp` parser is used to parse non-associative operators. Just like the former two parsers it takes parsers *p* and *op* as input and parses sentences of the form `p (op p)?`. If the sentence has the format `p op p op p`, where the *op* parser succeeds twice, an error is produced indicating that non-associative operators can not be used in the same sentence.

2.2.3 AST representation

The AST is constructed so that we have one element type for each of the classes "function declaration", "variable declaration", "statements" and "expressions". These element types hold all their child elements just like in the grammar. Due to the nature of the AST and the inherent precedence clarity we do not make a distinction between the different types of expressions in terms of Haskell types, they all have the expression type. We also store the location information in terms of start and end location of the overall element.

A valid SPL program can contain a list of variable and function declarations. The parser emits an `ASTUnordered` syntax tree which represents the global variable and function declarations in the order declared in the source file. Since in the later phase these functions declarations have to

be reordered according to their call dependencies, an `ASTOrdered` constructor also exists which stores the strongly connected components in terms of the call graph.

```
data AST =
    ASTUnordered [Either ASTVarDecl ASTFunDecl]
    | ASTOrdered [ASTVarDecl] [SCC ASTFunDecl]
```

Types are represented using the ASTType sum type which captures all the possible SPL types. In the case that a type is omitted in the source file, the special `ASTUnknownType` constructor is used.

```
data ASTType =
        ASTUnknownType EntityLoc
    |   ASTFunType EntityLoc [ASTType] ASTType
    |   ASTTupleType EntityLoc ASTType ASTType
    |   ASTListType EntityLoc ASTType
    |   ASTVarType EntityLoc Text
    |   ASTIntType EntityLoc
    |   ASTBoolType EntityLoc
    |   ASTCharType EntityLoc
    |   ASTVoidType EntityLoc
```

Variable declarations contain a variable name and an expression that determines the initial value. They potentially specify the type of the variable as well.

```
    data ASTVarDecl = ASTVarDecl EntityLoc ASTType ASTIdentifier ASTExpr
```

Function declarations specify the name and the names of the arguments. They also include the function body, which is a list of variable declarations and a list of statements.

```
data ASTFunDecl =
    ASTFunDecl
        EntityLoc
        ASTIdentifier
        [ASTIdentifier]
        ASTType
        ASTFunBody

data ASTFunBody = ASTFunBody EntityLoc [ASTStmt]
```

Statements can be control flow operations like if-else, while loops and return statements, but can also be variable assignments or function calls.

```
data ASTStmt =
    IfElseStmt EntityLoc ASTExpr [ASTStmt] [ASTStmt]
|   WhileStmt EntityLoc ASTExpr [ASTStmt]
|   VarDeclStmt ASTVarDecl
|   AssignStmt EntityLoc ASTIdentifier [Field] ASTExpr
|   FunCallStmt EntityLoc ASTFunCall
|   ReturnStmt EntityLoc (Maybe ASTExpr)
```

Expressions are used to represent conditions in control flow statements, new values of variables, function expressions, function arguments and return values. They can either be literal values, identifiers, function calls, field selections, tuples or expressions consisting of a unary or a binary operator.

```
data ASTExpr =
        IntExpr EntityLoc Integer
    |   CharExpr EntityLoc Char
    |   BoolExpr EntityLoc Bool
    |   EmptyListExpr EntityLoc
    |   EmptyCharListExpr EntityLoc
    |   IdentifierExpr ASTIdentifier
    |   FunCallExpr ASTFunCall
    |   FieldSelectExpr EntityLoc ASTExpr [Field]
    |   TupExpr EntityLoc ASTExpr ASTExpr
    |   OpExpr EntityLoc OpUnary ASTExpr
    |   Op2Expr EntityLoc ASTExpr OpBin ASTExpr
```

Since we support higher-order functions, the left-hand side of a function call does not have to be a literal function identifier. Instead it is an expression itself that should evaluate to a callable object. The arguments of a function call are represented as a list of expressions.

```
data ASTFunCall = ASTFunCall EntityLoc ASTExpr [ASTExpr]
```

2.2.4 Location information

In order to simplify location passing and computations a type-class Locatable is introduced. All AST components implement this type-class. It allows getting the start and end location of a component or of a list of components. The token datatype that is produced by the lexer also implements this type-class to make location handling uniform in the parser and independent of the source of the location information.

As seen in the structure of the AST, all elements contain an additional `EntityLoc` field which stores the location of that element in the source file. The `EntityLoc` is defined as:

```
-- Location is (LineNum, ColumnNum)
type Location = (Int, Int)

data EntityLoc = EntityLoc {
    locStart :: Location,
    locEnd :: Location
}
```

2.2.5 Parser Combinators

Our parser is a top-down, applicative parser that uses parser combinators to parse the tokens into an AST format. We have avoided using the monad instance of our parser since that would make the parsing process slower. We have opted to using parser combinators for this stage instead of parser generators for a number of reasons. Firstly we believe the readability of the code is greatly increased as the actual grammar of the SPL language can be easily deduced from the source code. This is because the parser combinators are abstract and high-level enough to make them readable. Additionally, we were intrigued by the simplicity of the solution and it inspired

us to create our own version of the standard parser combinator that would specifically suit our needs. Alternatively we could have used industry-level parser libraries such as `megaparsec` or `parsec` for this purpose but we have chosen not to do so. This is because these are fully-fledged libraries, so the learning curve is similar to the learning curve needed to implement our own parser minus the enjoyment of doing it yourself.

The Parser data type can be seen below. It can be viewed as a state-like function that takes a state and returns a list of possible parsing results and errors. The ParserState data type holds the current list of tokens together with statistical information such as how many tokens have been parsed. The Error data type holds an error of type 'e' and the depth at which the error occurred. The depth is taken as the number of tokens parsed at the moment where the error occurred. This information useful to the parser as it can selectively choose which error messages to keep during the parsing stage by comparing errors using their depth value.

```
newtype Parser s e a = Parser {
    runParser :: ParserState s -> [Either (Error e) (a, ParserState s)]
}
```

The applicative instance for the parser is rather standard (for each parsed result propagate this state to the next computation) and only differs in the way errors handled. More specifically, errors at each application of the sequential application operator (`<*>`) are removed if there happens to exist an error that is more favorable (higher depth). Additionally, error filtering happens in the strict version of the alternative operator (`<<|>`) which specifies that a parsing computation needs to take place only if the previous parsing computation fails.

Lastly, to improve the performance of the parser, all parser combinators have been written with left and right factoring in mind. This means that a parser parsing sentences of the form `p1 p2 <|> p1 p3` will be left refactored to `p1 (p2 <|> p3)`, and similarly a parser taking sentences of the form `p1 p2 <|> p3 p2` will be right factored to `(p1 <|> p3) p2`. Using this optimization we can parse heavily nested comments and tuples efficiently without hanging the compiler.

A downside of our parser is that it fails on the first error when there are no valid parsing branches. This means that our parser will display at most one parse error to the user, the error produced at the largest parse branch.

2.3 Combined Frontend

The frontend of the compiler consists of both, the lexer and the parser. The lexer transforms the code into a list of specified tokens if that is possible. This list is passed to the parser and turned into an AST representation of the program syntax if the list of tokens forms a syntactically correct program of SPL. If the lexing step fails no list of tokens is produced, so the parsing step cannot be executed. If the lexing step succeeds, it is not guaranteed, that the parsing step is also successful. The following example contains valid tokens only, but it does not form a syntactically correct program. It does not produce any errors in the lexing step, but the parsing step fails.

```
True var = a - 'c';
```

Chapter 3

Analyses & Typing

In the Semantic Analysis phase the compiler performs a series of consistency checks to ensure the code is in a semantically valid format. Additionally, overloading is resolved using implicit arguments, and optimizations such as constant folding and dead code elimination also occur in this phase.

For our compiler we have implemented six sub-phases which are as follows. We note that the order of these sub-phases listed here is also the order by which we execute them.

1. Identification of Mutually Recursive Functions (Section 3.2)
2. Constant Initialisation of Global Variables Check (Section 3.2)
3. Polymorphic Type Inference and Checking (Section 3.2)
4. Return Path Check (Section 3.5)
5. Constant Folding and Dead Code Elimination (Section 3.6)
6. Overloading Resolution (Section 3.4)

3.1 Core Language

We use a new kind of abstract syntax tree for type checking and for performing semantic analysis, which we name the Core Language.

There are multiple reasons for the introduction of a Core Language at this stage. Firstly, the Core Language represents a semantically valid program where all type information have been inferred and are correct. In the AST, types can be unknown (e.g. declared using `var` keyword) or not specified at all as in the case of unspecified function types. In the Core Language, all missing types have been inferred thus every variable declaration, function declaration or function call expression contains the actual inferred type of the entity. Additionally, inferred type information are also stored in function calls, in assignment statements, in expressions storing function identifiers, and lastly in binary operators.

Secondly, the Core Language represents a minimalistic representation of the program thus features such as field selector expressions are translated to nested function calls. As a result, the semantic analysis stage of the compiler can be kept as lean as possible without suffering from seemingly duplicate code.

Lastly, the introduction of a dedicated structure allows for quick changes to the Core Language without having to change other non-related parts such as AST parsers. When a change to the Core Language is deemed desirable we need to change the data structure itself and the generator that converts an AST tree to the Core Language. There is no need to adapt the AST parsers to encompass this change thus introducing an independence between the two phases.

3.2 Polymorphic Type Inference and Checking

Our type checker is best described as a let-polymorphic type inference and checking procedure. In this section, we elaborate on the key concepts of our type checker together with its features and limitations.

Inspired by Grabmuller's paper "Algorithm W Step by Step", we opted to represent universally quantified types using the notion of type schemes, together with the associated generalisation and instantiation procedures to lift a type to a scheme and transform a scheme to a type respectively. Type schemes can only be stored in the type environment that holds the declared functions and variables in scope. After the most general type of a function or variable is inferred, this type is compared with the possibly supplied user specified type of the entity and it is only accepted if the user supplied type is an instance of the most general type. In the case of functions, the resulting inferred/user-supplied type is generalised by quantifying over the type variables in the type that do not appear in the type environment. Variables on the other hand are lifted to type schemes and added to the type environment without any generalisation occurring. Additionally, when a function is used in a function call, the type scheme of the function is retrieved from the type environment and it is instantiated to a standard type.

3.2.1 Type Environment

As described above, the type environment holds the type schemes of the functions and variables. Functions can contain universally quantified type variables whereas variables do not. Scoping rules are taken into account by hiding global variables/functions in the presence of argument or local variables with the same name. Similarly, local variables will hide argument variables with the same name. When a function or variable is not found in scope the type checker produces a user-friendly error.

A type environment is created incrementally, by type checking variables and functions in an ordered fashion, generalising them, and adding them to the environment. To support access of a function before definition and mutually recursive functions, the top level statements in the code are reordered and placed into groups of mutually recursive functions. The reordering of statements happens by generating a call graph of the program and supplying this call graph to an algorithm that returns a topologically sorted list of the strongly connected components (SCC) of the graph. The library used to perform this kind of analysis implements a functional algorithm for identifying SCC of a graph [KL95]. This topologically sorted list contains the function declarations in a sorted order so that function calls in expressions refer to function declarations that are ordered before the corresponding expression. When two or more functions are mutually recursive though, the algorithm will correctly identify them as a strongly connected component, and the type checking procedure will take this into account by firstly adding a fresh type for each function in the type environment, then by type checking them simultaneously and only afterwards generalising all the mutually recursive functions in the component.

3.2.2 Treatment of Global Variables

We note that we only allow mutually recursive functions and not mutually recursive variable declarations. Firstly, global variables may only be initialised using constant expressions i.e. expressions that contain no function calls or accesses to other variables. Thus mutual recursion is disallowed in global variables by requiring them to be initialised in a constant way. Local variables in a function definition do allow function calls and other variable accesses but the order of a variable definition matters. A local variable definition that accesses another local variable that is declared later on will produce a variable not in scope error. We consider that such restrictions are reasonable since it is not intuitively clear how mutually recursive variable definitions should be handled.

3.2.3 Support for Higher-Order Functions

In order to account for partial application of functions, types in the Core Language are always represented in their curried form. This representation simplifies the unification process as unifying the partially applied form of a function type with the actual function type can be done with no special case distinction.

For instance, if function types were represented using the SPL syntax of $Type* \rightarrow Type$, the expression $add(5)$ would generate a unification step between types $Int\ Int \rightarrow Int$ and $Int \rightarrow a$ for some fresh variable a. Here, a should be substituted with the type $Int \rightarrow Int$, but for the unification step to work, the actual function type would need to be translated to its curried form first. With the curried function type representation, there is no need for this special case treatment.

The information of how much arguments a function takes is not lost, as it can still be derived from the function declaration. As a result, the SSM code generation can still use these information to optimize away an extra thunk creation and call a function directly if it is passed all its arguments.

```
data CoreType =
        CoreIntType EntityLoc
    |   CoreBoolType EntityLoc
    |   CoreCharType EntityLoc
    |   CoreVoidType EntityLoc
    |   CoreVarType EntityLoc TypeVar
    |   CoreTupleType EntityLoc CoreType CoreType
    |   CoreListType EntityLoc CoreType
    |   CoreFunType EntityLoc (Maybe CoreType) CoreType
```

As seen from the Core Language type representation, a function type takes a `Maybe CoreType` and returns another `CoreType` back. The `Maybe` wrapper is needed to represent functions that take no arguments. This means that the type system treats functions that take no arguments differently from values of the same result type.

3.2.4 Recursion

The approach we have taken in handling recursive definitions is using a monomorphic typing rule. Thus a recursive function can only call itself inside its body with the same argument types that it has been called with. The converse of this approach is polymorphic recursion which could be used if a recursive function has been given a user-supplied type. We do not support polymorphic recursion as certain design decisions complicate the implementation.

3.2.5 Global Substitution Context

To increase code readability and robustness, we accumulate inferred substitutions from the unification process into a global substitution context. This substitution is implicitly passed via a State monad as seen below:

```
data TypeCheckState =
    TypeCheckState {
        globalSubst :: Substitution,
        env :: TypeEnv,
        ...
    }

type TCMonad a = StateT TypeCheckState (Either Error) a
```

When a new substitution constraint is generated, it is added to the substitution context, and the environment is updated accordingly. Since only during unification do types need to be substituted to their most concrete form, we can defer applying substitutions to all the types in scope. This increases code readability as the code is not littered with substitutions and the code is more robust as there is no need to worry for a missing substitution.

3.2.6 User Specified Types

A design decision we have opted to employ is that type variables occurring in user-specified types are always unique and name clashes with other type variables in the type environment are resolved by renaming and not by linking them. This design decision is similar to Haskell's default way of treating user-specified types with the alternative being to be able to refer to type variables in a function type declaration inside the function body itself. Such behavior can only be enabled using the language extension *ScopedTypeVariables* that links type variables with the same name in different user-supplied types together.

3.2.7 Error Reporting

Support for user friendly type checking error messages is done by linking each type to a node in the Core Language. When a type mismatch between expected and actual type occurs, the type will contain the exact location of the element in the source code that this type links to, thus an error message showing which variable or function has an invalid type is produced. Additionally, auto-generated type variables may contain cryptic names, thus we often attempt to link auto-generated type variables with the corresponding type variables that have been supplied by the user in function and variable type declarations.

3.3 The Void Type

In order to avoid complicating the type checker with extra conditions to treat the *Void* type as a special type that cannot be assigned to variables and function arguments, we have instead opted to treat the Void type as simply a unit type with just one inhabitant.

Similar to Python's *NoneType*, the Void-type in our type system can be assigned to variables and to function arguments just like any other type, while it can also be used for comparisons and for printing. As a result, a list or tuple of Void-type elements is perfectly accepted by the typechecker, and Void-type elements are always equal to each other. When a function is assigned

a Void-type in its return value, the code generator will generate a function that returns the single inhabitant of the Void-type thus in principle generated functions in the lower level representation will always return a value.

The only difference of the Void-type with all the other types in our system is that it can not be directly created by a literal expression but can instead only be created by a function that returns no result.

3.4 Overloading Resolution

Multiple ways of supporting overloading of operators and functions exist and in our compiler we have opted in taking the implicit argument-based approach.

In essence, after the type checking phase an overloading resolution phase proceeds, which consists of a gathering of type constraints step that infers all the overloaded type constraints of each function declaration and a rewriting step that rewrites all function calls to an overloaded function with the additional composed overloaded instances.

3.4.1 Gathering Step

The type constraints produced by the gathering step are requests for additional function arguments that point to certain implementation of an equality, comparison or printing method. There are seven groups of type constraints, namely Equal α, LessThan α, LessEqual α, GreaterThan α, GreaterEqual α and Print α where α stands for the type variable being constrained. For instance, assuming $x, y : a$ are arguments of a function, if during the gathering step an expression such as:

```
('d', x) == ('e', y)
```

is encountered, then the Equal a type constraint will be produced which indicates that a function to compare inhabitants of type a is needed for the overloaded expression to work. Thus the gathering step infers the type constraints of type variables occurring in a function type. There are two places where type constraints may be produced which are in expressions involving equality or comparison operators, or in function calls to overloaded functions such as the built-in `print` function and other inferred overloaded user functions.

In the case of mutually recursive functions, the gathering step has to be repeated with each iteration updating the function environment with the new overloaded type constraints inferred. This is needed since at every iteration the type constraints inferred for a function in the mutually recusive block might increase as the type constraints propagate to the other functions in the block.

Take for instance the following three functions:

```
f(x,y,z) :: a b c -> a {
    print(x):
    g(x,y,z);
    return x;
}
```

```
g(x,y,z) :: a b c -> b {
    print(y);
    h(x,y,z);
    return y;
}
```

```
h(x,y,z) :: a b c -> c {
    print(z);
    f(x,y,z);
    return z;
}
```

In the beginning, functions f, g, h have no overloaded type constraints stored in the environment. As a result, after the first gathering iteration, function f will have a $Print\ a$, function g will have a $Print\ b$ and h will have a $Print\ c$. At this moment, the function environment is updated to store the new type constraints inferred. But the inferred type constraints are not complete, as all three functions must have $Print$ constraints for all the type variables. Thus, a

new gathering iteration follows which results in function f having $Print\ a$ and $Print\ b$, function g having $Print\ b$ and $Print\ c$ and function h having $Print\ c$ and $Print\ a$. This iteration does not result in the complete set of type constraints yet, and another last iteration is needed to infer all the constraints.

In essence, the gathering step for mutually recursive functions must be repeated up until no new type constraints are generated, with the size of block yielding the maximum number of iterations.

3.4.2 Rewriting Step

In the rewriting step, new function arguments are generated that correspond to the instances of the inferred type constraints and expressions are rewritten by replacing them with function calls or by appending extra arguments to existing function calls. The composition of needed overloaded instances also happens at this step.

In the case of overloaded operators, the type of the operator gives us the needed overloaded instance. For instance, assuming again that $x, y : a$ are function arguments, an expression such as $('d', x) == ('e', y)$ requires a way to compare tuples of type $(Char, a)$. This equality instance is composed on the spot from the following higher-order instance combinator:

```
_equal_tuple :: (a a -> Bool) (b b -> Bool) (a,b) (a,b) -> Bool
```

which takes as input a way to compare inhabitants of type a and b and returns a function to compare tuples of type (a, b). Using this combinator, we can compose the needed instance for type $(Char, a)$ using the expression `_equal_tuple(_equal_char, _equal_a)`. Here, `_equal_char` represents the function to compare characters (generated in the SSM backend) and `_equal_a` refers to the function argument for the constraint $Print\ a$. At this point the overloaded operator is replaced with a call to the generated expression.

For overloaded functions, a similar rewriting step occurs where references to overloaded function declarations occurring inside a function body are replaced with function calls that pass the overloaded instances to the function. If the function identifier is occurring in a function call expression, the whole function call is rewritten by prepending it with the extra instances instead of composing function calls together. This is a slight optimization which saves us from one extra thunk being created and evaluated.

3.4.3 Restrictions

There are two restrictions to overloading that take place, firstly for ambiguous overloaded instances, and secondly for overloaded no-argument functions.

Firstly, we disallow overloading of functions if the type variable referenced in the type constraints is ambiguous from context. For instance, the following program will throw an ambiguous type constraint error, since the $Print\ a$ in the list xs cannot be resolved to an actual instance.

```
foo() :: -> Void {
    [a] xs = [];
    print(xs);
    return;
}
```

Secondly, functions that take no arguments cannot be overloaded, as the rewriting step will also evaluate the function. Take for instance the following program:

```
foo() :: -> [a] {
    [a] xs = [];
    print(xs);
    return xs;
}
main() {
    (-> [Int]) foo2 = foo;
}
```

The rewrite step for this program will replace the identifier foo in the main function with the expression $foo(\_print\_int)$. But this rewrite step introduces the unintentional side-effect of evaluating the function as well, and thus $foo2$ is no longer a function but a list of type $[Int]$. For this reason, an error is produced when overloading a function that takes no arguments. We note that it is perfectly fine for foo to call overloaded functions/operators in itself as long as these type constraints can be resolved inside its body thus not overloading itself.

3.5 Return Path Analysis

All return statements are already guaranteed to have the correct type by the type inference described in section 3.2. The remaining check only needs to make sure that any function that returns a type other than Void does so in every execution branch produced by if-else statements and while loops. In the following explanation only functions returning a non Void type are considered.

A list of statements is guaranteed to return if any of the contained statements does so. Assign statements do not return, while return statements obviously do. The interesting cases are statements that contain lists of statements themselves such as if-else and while statements. In general it is possible that the body of a while loop is never executed, so even if there is a return statement within the body of a while loop, the while loop is never guaranteed to return. If-else statements that only return in one of their branches are not guaranteed to return. However, if-else statements, that return in both of their branches, are guaranteed to return.

The fact that a function returns is determined by the list of statements within its body. If and only if this list of statements is guaranteed to return, the function is guaranteed to return a value as well.

3.6 Constant Folding and Dead Code Elimination

This optimization phase performs constant folding and dead code elimination. Thus the goal is to identify static expressions, evaluate them at compile time and possibly eliminate dead code that cannot be reached. We note that constant propagation does not occur since we do not support constant variables but only constant expressions. The obvious advantages of such optimization is that unnecessary re-computations do not happen, and the binary size is smaller.

Different types of expressions are evaluated at this stage which take into consideration the short-circuit behavior of logical operators, and of certain facts about equality that are known to hold. For instance, the following expressions in the left-hand side of the arrow are optimized to their minimal form shown on the right-hand side.

```
5 + 5 |-> 10,  5 > 8 |-> False

False && someExpr |-> False,  True || someExpr |-> True
```

```
x == x |-> True,   x != x |-> False
```

After all expressions are optimized as described, dead statements in function definitions are eliminated. For instance, if-else statements with a constant True or False condition expression are replaced with empty taken or not taken branches, while statements with a constant False condition expression are removed and lastly statements after a return statement are removed.

Chapter 4

Code Generation

In the Code Generation phase, the compiler generates SSM assembly code from the Core language. This assembly code can be executed using the SSM interpreter. In this section, we discuss the design of the SSM backend. We note that initially we implemented an Intermediate Representation for SPL that the Core language first compiled to but that has been removed as it provided no real benefits.

4.1 Expressions

Expressions need to be broken up into atomic operations as the SSM can only perform one operation at a time. An SPL expression can be directly translated to a list of SSM instructions by simply observing that evaluation of a single expression always results to one extra stack cell – the result of the expression. For example, translating the expression $3 * (1 + 2)$ happens by firstly loading the constant 3, and then translating the expression $1 + 2$. Multiplying the two stack cells now results in one new stack cell in total.

```
load constant 3
load constant 1
load constant 2
add
mult
```

4.2 Assignments

Assignments can be made to local and global variables, but they both share the same principle. Each variable lives at a certain memory address. In case of local variables, this memory address is inside the stack frame of the current function. In case of global variables, this memory address is inside the heap pointed to by the special GP register Assignments assign the result of an expression to this memory address. For instance, $x = 3 + 1$ translates to SSM code like this:

```
load constant 3
load constant 1
add
load memory address of var x
sta 0 # store second value on stack to memory address on top of stack
```

4.2.1 Function Calling Conventions

We have opted in using call by value conventions for basic types such as integers, and characters and call by reference for composite types such as lists and tuples similar to the C/CPP calling style.

During a function call, every argument gets copied to the stack by the caller. For basic types, this corresponds to a call-by-value convention. Composite types on the other hand are represented as pointers (memory addresses) to the objects stored on the heap, thus only the pointer to the structure is copied.

Moreover, the caller is given the responsibility of popping the arguments from the stack after the function call has returned instead of delegating the responsibility to the callee.

4.2.2 Polymorphism

Polymorphic functions are treated as normal functions taking arguments of some unknown type. This is not a problem as all data types have the size of one stack cell.

4.2.3 Lists and Tuples

All composite data structures are stored on the heap. A list takes two words, the first word stores the pointer to the tail of the list and the second word stores the actual element. A pointer to a list will always point to the second word (the element) of the list cell. The empty list is not stored directly in the memory, but instead it is represented by a pointer of value *0*. Similarly, a tuple needs two words in size. The first word stores the first element and the second word the second element. This representation is sufficient for more complex structures, as any structure taking more than one word in size is always represented as a pointer to it. For instance, a structure of type `[(Int,Int)]` uses two words, a pointer to a tuple and a pointer to the tail of the list.

4.2.4 Overloading

As discussed in previous sections, the compiler supports overloading of equality, comparison and printing operators/functions. All kinds of composite structures can be compared and printed using the canonical way of comparing and printing their sub-components.

In the Semantic Analysis stage, the compiler inferred for each function the requested implementations of overloaded operators/functions and has rewritten every access to an overloaded function or operator. Thus, at the SSM generation stage, overloading has already been resolved and only the primitive instances need to be defined.

The instances defined at the SSM level are equality, less-than, less-equal, greater-than, greater-equal and printing of the basic types Int, $Char$, $Bool$ and $Void$. Additionally, the instance combinators for lists and tuples are also defined.

4.3 Runtime System

This section expands on the run-time aspect of higher-order functions. We note that support for higher-order functions in the syntax has been described in Section 1.2.6 and Section 1.2.7 and support by the type checker has been described in Section 3.2.3.

Partial application of functions requires thunks to store the function and the arguments that are given to the function. Thunks are passed between functions, so they need to be stored on the heap. Two subroutines are needed for handling thunks, one to store and one to call thunks. The following example demonstrates the functionality of both subroutines:

```
add(x, y, z)
{
    return x + y + z;
}

main()
{
    var a = add(3, 4);
    var b = a(5);
    print(b);
}
```

The representation of the thunk for variable "a" looks like this:

```
|-----  <bottom of stack>
|  4    <- arg2
|-----
|  3    <- arg1
|-----
| add   <- function label
|-----
|  3    <- number of arguments the function takes
|-----
|  2    <- number of arguments provided in the thunk
|-----  <top of stack>
```

The subroutine that stores thunks on the heap expects exactly this structure on the stack before it is called. The structure is then replicated into heap and the pointer to the last element is returned. If the thunk is now called using "a(5)" in this case, the following structure is produced on stack:

```
|-----  <bottom of stack>
|  3    <- arg1 of thunk call
|-----
|  a    <- address of thunk (points to last element)
|-----
|  1    <- number of new arguments given for the thunk call
|-----  <top of stack>
```

Calling the subroutine for calling thunks then puts the new argument on stack and loads the thunk afterwards. The number of arguments provided in the thunk is then increased by the number of new arguments provided by the thunk call. The resulting structure on stack looks like this then:

```
|-----  <bottom of stack>
|  5    <- arg3 (new arg1 given by the thunk call)
|-----
|  4    <- arg2 (from thunk load)
|-----
|  3    <- arg1 (from thunk load)
|-----
| add   <- function label
|-----
|  3    <- number of arguments the function takes
|-----
|  3    <- number of arguments provided in the thunk (was increased by 1)
|-----  <top of stack>
```

The subroutine then compares the number of arguments the function takes to the number of arguments given in the thunk (this is done without destroying the stack structure by working with copies of the values). If the two numbers are equal, like in this case, the function can be evaluated. To do that, two elements are popped from the stack and the function given in the thunk is called using the "jsr" instruction. The result of the function call then resides in the return register so the subroutine can return and provides the result of the function call. If the

number of arguments given in the thunk is less than the number of arguments required, the store thunk routine is called to copy the new thunk structure to heap. The address of this new thunk is then in the return register and the routine for calling thunks can return and provides the address of the new thunk on heap.

The special structure of thunks with inverted order of arguments was chosen carefully as it allows for easy handling like shown in the example. If a function is called and all arguments are provided, we make the function call directly and avoid the creation of a thunk. On the other hand, we create a thunk for every function that is passed into another function as an argument. We avoid making the distinction of calling a thunk or a function label that is passed as an argument to a function.

Appendix A

Grammar

The modified grammar used for the SPL language can be seen below. Here expressions are suffixed with an L to represent left associative rules and N for non associative rules and R for right associative rules.

```
SPL          = Decl+
Decl         = VarDecl | FunDecl
VarDecl      = ('var' | Type) id  '=' Exp ';'
FunDecl      = id '(' [ FArgs ] ')' [ '::' FunType ] '{' Stmt+ '}'
Type         = FunType
             | BaseType
FunType      = BaseType* '->' BaseType
BaseType     = 'Int'
             | 'Bool'
             | 'Char'
             | 'Void'
             | '(' Type ')'
             | '(' Type ',' Type ')'
             | '[' Type ']'
             | id
FArgs        = [ FArgs ',' ] id
Stmt         = 'if' '(' Exp ')' Stmt [ 'else' Stmt ]
             | 'if' '(' Exp ')' '{' Stmt* '}' [ 'else' '{' Stmt* '}' ]
             | 'while' '(' Exp ')' Stmt
             | 'while' '(' Exp ')' '{' Stmt* '}'
             | VarDecl ';'
             | id Field '=' Exp ';'
             | FunCall ';'
             | 'return' [ Exp ] ';'
Exp(L)          = LAndExpr ( '||' LAndExpr )*
LAndExpr(L)     = EqExpr ( '&&' EqExpr )*
EqExpr(N)       = LeGrExpr [ ('==' | '!=') LeGrExpr ]
LeGrExpr(N)     = ConsExpr [ ('<=' | '>=' | '<' | '>') ConsExpr ]
ConsExpr(R)     = AddSubExpr ( ':' AddSubExpr )*
AddSubExpr(L)   = MultDivExpr ( ( '+' | '-' ) MultDivExpr )*
MultDivExpr(L) = UnOpExpr ( ( '*' | '/' | '^' | '%' ) UnOpExpr )*
```

```
UnOpNotExpr(R) = '!' UnOpNotExpr | UnOpNegExpr
UnOpNegExpr(R) = '-' UnOpNegExpr | BasicExpr
BasicExpr   = '(' Exp ')'
            | Tuple
            | FieldSelector
            | FunCall
            | String
            | List
            | id
            | int
            | char
            | 'False'
            | 'True'
String        = '"' .* '"'
List          = '[' ZeroOrMoreExp ']' [ Field* ]
Tuple         = '(' Exp ',' Exp ')' [ Field* ]
FieldSelector = id Field+
              | '(' Exp ')' Field+
Field         = '.' 'hd' | '.' 'tl' | '.' 'fst' | '.' 'snd'
FunCall       = id '(' ZeroOrMoreExp ')'
              | '(' Exp ')' '(' ZeroOrMoreExp ')'
ZeroOrMoreExp = ( [ ( Exp ',' )* ] Exp )?
int           = digit+
id            = alpha ( '_' | alphaNum)*
```

Bibliography

[KL95] David J. King and John Launchbury. Structuring depth-first search algorithms in haskell. pages 344–354. ACM Press, 1995.