Introducing Typeling: an Imperative Programming Language with Algebraic Data Types

VICTOR HORNEȚ, University of Twente, The Netherlands

Supervisor: Peter Lammich

Algebraic data types (ADTs) allow easy modelling of complex data structures. When paired with static type checking, they can empower the programmer to write more robust and reliable code. However, ADTs are not as popular in the context of imperative programming languages, compared to functional ones. This research paper investigates the current use of algebraic data types and proposes a new programming language that aims to make ADTs more accessible to beginner programmers with an imperative background. We explore existing implementations of ADTs in popular languages and combine their best features into the design of a new programming language called "Typeling", then we construct a prototype compiler for it using the LLVM compiler infrastructure.

Additional Key Words and Phrases: programming language design, algebraic data types, LLVM, prototype construction

1 INTRODUCTION

Algebraic data types (ADTs) are a kind of composite types characterized by combining other types through the use of "algebraic" operations. The two most common classes of ADTs are product types and sum types. Product types represent a collection of values that are all needed together. Examples of product types include tuples and records. On the other hand, sum types (i.e., tagged or disjoint unions or variant types) represent a value that can be one out of a set of multiple choices. ADTs have several benefits in programming. One of their greatest strengths is that they allow for the intuitive modelling of complex data structures, which improves code readability. Furthermore, by modelling the application's business logic with ADTs and ensuring that illegal states are not represented, one can make their code more robust.

Algebraic data types have a strong connection with the functional programming paradigm. One of the first languages to introduce them is HOPE [11], a functional programming language whose goal was to "encourage the construction of clear and manipulable programs." However, ADTs are less commonly used in imperative programming languages. Therefore, we established the following objectives for the research project:

- To design a beginner-friendly imperative programming language which has **algebraic data types** as the main feature.
- To use the **LLVM Compiler Infrastructure** to implement a prototype compiler for the resulting language specification.

As a result, the research questions listed below have been posed:

RQ1 How are ADTs currently used in practice by modern programming languages?

RQ2 How can our new programming language implement ADTs using the LLVM IR?

RQ3 Which features could be added to the programming language to improve the usability of our ADTs implementation?

In this paper, we first defined the requirements that our new language design must adhere to, as seen in Section 2. Then, in Section 3 we answered the first research question (**RQ1**) by investigating how some of the most popular programming languages currently use ADTs in practice. Next, in Section 4, we presented the design and example code of our new programming language, Typeling, which follows the requirements formerly specified in Section 2. In addition, we built a prototype JIT compiler for our design using the Rust programming language and the LLVM compiler infrastructure and use its implementation to answer **RQ2**. Following that, we applied our findings from Section 3 to reflect on Typeling's strengths and weaknesses, and to propose expansions to our design that would solve some of the current weaknesses. We finalize the paper by discussing potential further work on the Typeling compiler.

2 DESIRABLE LANGUAGE REQUIREMENTS

Before reviewing the currently existing solutions, we first defined the desired requirements for our language. This section goes into the selected requirements and motivates their choice.

2.1 Simple syntax

The syntax should be concise and unambiguous to ensure it is easy to read and write. Reserved keywords should be as short as possible, no longer than six letters. Furthermore, reserved symbols should only be chosen if their meaning can be extracted from their context.

2.2 Static typing

Statically typed languages with a proper type checker allow programmers to avoid having to deal with the "inscrutable bugs" which come with dynamic typing [18]. Therefore, we consider static typing an essential requirement for the language. However, they should be implemented unobtrusively, so the user could avoid writing them as much as possible.

2.3 User-defined types

The main feature of the language should be its type system. According to Burstall et al. [11]., the availability of a simple and powerful type-definition facility dramatically simplifies the programmer's tasks. Therefore, the language should have an ADT-based type system to encourage the users to implement their own types as much as possible. One primitive type, a 64-bit integer, should be provided as a base for building more complex types. Lastly, some form of pattern matching should be implemented as well to allow the user to access the ADTs' inner data.

TScIT 39, July 7, 2023, Enschede, The Netherlands

© 2023 University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

2.4 Imperative control flow

According to a public survey [20], the majority of people who are learning to code choose an imperative programming language as their first one. As such, it makes sense for the language to feature sequential composition, and to have the following control flow structures:

2.4.1 Functions. The language should implement functions to allow for further modularity of the code. Moreover, it should also be possible to write recursive functions to permit the traversal of nested ADTs.

2.4.2 While loops. Loops are an essential feature of imperative programming languages. They empower the user to perform repetitive tasks, as well as iterate over data structures. Even though the same effect could be achieved with recursive functions, the language should also have `while` loops to promote the imperative style.

2.4.3 If statements. Conditional branching in the form of `if` statements is an essential control flow structure of imperative programming languages, therefore it should be supported by the language.

3 EXISTING SOLUTIONS

Functional programming languages were the first to introduce ADTs, being implemented by the HOPE programming language [11]. It makes sense, then, that functional programming languages are the ones that most commonly use ADTs. Among the most well-known ones is Haskell [17] and the ML [18] family of languages, which include Standard ML [19] and OCaml [4, 16].

However, our focus is on implementing an imperative language, therefore we have reviewed some of the most popular imperative programming languages according to [20], namely *JavaScript* [3], *Python* [5], *Java* [1], *C++* [14], and *Rust* [6, 7].

3.0.1 JavaScript. A multi-paradigm language that supports both functional and imperative programming styles. JavaScript [3] is dynamically typed, therefore it does not adhere to our requirements. It also has a reputation for being an unintuitive language because of some of its constructs, such as the difference between the "==" and "===" operators. Nevertheless, in 2012, Microsoft released a superset of JavaScript, namely TypeScript [8]. This release adds optional static typing to JavaScript, making it easier to write and maintain large-scale applications. TypeScript allows the construction of ADTs as well, through its union ("|") and intersection ("&") operators. However, it does not support pattern matching.

3.0.2 Python. Another multi-paradigm language that supports both functional and imperative programming styles. Like JavaScript, Python is dynamically typed, so it does not fit our criteria. Python became a popular language due to its simplicity, readability, and many libraries and frameworks. Python only started supporting structural pattern matching and the expression of algebraic data types in version 3.10 [10], released in 2021.

3.0.3 Java. An imperative language that is statically typed. Java is a popular language for enterprise development due to its scalability, security, and cross-platform compatibility. It has a large and active community, which has led to a vast ecosystem of libraries and frameworks. As of Java 16 [1], pattern matching has become a standard feature, which allows the implementation of ADTs through a combination of inheritance and `instanceof` expressions. However, this method can get quite verbose. Nevertheless, there have been attempts at implementing more succinct ADTs and pattern matching by libraries such as Spotify's DataEnum [2].

3.0.4 C++. An imperative language that is statically typed. C++ [14] is a popular language for systems programming, game development, and high-performance computing due to its speed, efficiency, and low-level control. While C++ does not have built-in language support for ADTs, similar functionality can be achieved by combining classes, inheritance, and variants. However, C++ does not provide native pattern-matching syntax.

3.0.5 Rust. A multi-paradigm language that supports both functional and imperative programming styles. Rust [6, 7] is a statically typed language, first released in 2010 by Mozilla Research. Rust features ADTs and pattern matching as first-class language features. However, it can be difficult to learn due to its strict compiler, and the safety guarantees provided by the borrow checker can sometimes be limiting. Rust served as one of the main inspirations for the Typeling language.

4 THE TYPELING PROGRAMMING LANGUAGE

We designed the Typeling programming language according to the requirements specified in Section 2. As formerly stated, our goal was to create a simple programming language that would enable beginners with an imperative programming language background to become acquainted with algebraic data types.

In its current state, Typeling is still a prototype language. It is not intended to be a revolutionary technological breakthrough, but rather an experiment in language design. Therefore, it has some limitations, such as a lack of input facilities and reliance on the `printf` C library function for output.

4.1 Comments

Typeling features C-style comments. Single-line comments must be prefixed by the "`//`" symbol and multi-line comments must be surrounded by the "`/*`" and "`*/`" symbols.

```
// single-line comment
/* mult-line
comment */
```

4.2 Type system

As stated in Section 2, the Typeling language only has one primitive type, namely a 64-bit signed integer. In Typeling, the reserved keyword for the 64-bit integer is "`i64`".

Similarly to the HOPE [11] and Haskell [17] languages, data types in Typeling are represented as *data constructors* which are applied to several terms, which in turn represent another data item.

A new type can be declared using the `type` keyword, followed by the name of the type and a list of data constructors. For example, to define a `Num` type that acts as a wrapper over an `i64`, one would write:

```
type Num = Num i64
```

which defines a new data type called **Num** with one *data constructor*, called `Num`, that takes an `i64` as an argument.

Note that types and their constructors are allowed to have different names, for example:

```
type Num = I i64
```

which as well creates a data type called **Num**, but whose constructor is `I` instead.

There are three kinds of data constructors, classified according to their parameter count and types:

4.2.1 Unit constructor. A *unit constructor* has no parameters. Figure 1 outlines two methods of declaring the same **Unit** type.

```
// standard  notation
type Unit = Unit

// shorthand notation
type Unit
```

Fig. 1. Declarations of a new type with no fields (Unit constructor)

Unit data structures can then be created by calling the *unit constructor* with no arguments:

```
x := Unit;
```

4.2.2 Tuple constructor. A *tuple constructor* can have multiple parameters. Tuple types can be declared in three ways, as seen in Figure 2.

```
// standard notation
type Tuple = Tuple i64 i64

// alternative notation
type Tuple = Tuple (i64, i64)

// shorthand notation
type Tuple(i64,i64)
```

Fig. 2. Declarations of a product type with anonymous fields (Tuple constructor)

Once defined, *tuple constructors* can be called with the appropriate number of arguments to construct its data structure:

```
x := Tuple(10,20);
```

4.2.3 Struct constructor. Just like *tuple constructors*, a *struct constructor* can have multiple parameters as well. However, the parameters of a *struct constructor* can be named, as shown in Figure 3.

```
// standard notation
type Struct = Struct x:i64 y:i64

// alternative notation
type Struct = Struct {x: i64, y: i64}

// shorthand notation
type Struct {x: i64, y: 64}
```

Fig. 3. Declarations of a product type with named fields (Struct constructor)

Again, like *tuple constructors*, *struct constructors* can be called with the appropriate number of arguments to create their data structure. However, they also allow the arguments to be provided in any order, by prefixing them with their field name[1]:

```
x := Struct(10,20);
y := Struct(y=20,x=10);
```

4.2.4 Sum types. Additionally, sum types (or enums) can be obtained by combining data constructors with the `"|"` symbol. An example is shown in Figure 4. Here, any of the calls to the `A`, `B`, `C` or `D` constructors will result in an `Enum` data type.

```
// standard notations
type Enum = A
          | B i64
          | C i64 i64
          | D x:i64 y:i64 z:i64

// alternative notations
type Enum = A
          | B (i64)
          | C (i64, i64)
          | D {x:i64, y:i64, z:i64}
```

Fig. 4. Declarations of a sum type

In addition, new type declarations can contain other types, by referencing the name of the target type as a subterm in the data constructor. For example, Figure 5 shows the declaration of a linked list of 64-bit integers.

```
type List = Cons i64 List | Nil
```

Fig. 5. Declaration of a List type

4.3 Functions

Typeling programs consist of multiple (global) type and function declarations. Functions are declared using the `"fn"` keyword and

[1] due to a bug, this feature is not currently supported by the prototype and will result in a type checking error

require a name, a list of parameters and a return type. The parameters of a function must be explicitly typed. For example, a function which computes the sum of two integers would have the following signature:

```
fn sum(x: i64, y:i64) -> i64 { /* statements */ }
```

If a function does not return a value, its "`-> type`" part can be omitted. Such functions return the "`()`" unit type. The "`main`" function is one such example. It serves as the entry point of a Typeling program. Therefore, programs missing the "`main`" function will fail to compile. Its signature is as follows:

```
fn main() { /* statements */ }
```

The order of function declarations does not matter. In the following example:

```
fn fun1() {}
fn fun2() {}
```

the "`fun1`" function will be able to call "`fun2`", even though it has been declared before the latter. It then follows that recursive calls are also possible.

4.4 Statements

As is the case with imperative languages, function bodies are created by the sequential composition of *statements*. In Typeling, a *statement* is an instruction that does not produce a value.

The Typeling language features three main control flow statements: conditional branching with `if` statements, `while` loops, and function `return` statements.

4.4.1 Function returns. The `return` statement can be used to terminate the execution of a function, returning the control flow to its caller. In addition, the `return` statement can be used to mark the value produced by the function. Therefore, functions which have a return type other than `()` must have a `return` statement, as such:

```
fn three() -> i64 {
    return 3;
}
```

For functions that do not produce a value, the `return` statement can be omitted, as it will be implicitly added as the last statement of the block. For example, this:

```
fn nothing() {}
```

is equivalent to:

```
fn nothing() { return; }
```

4.4.2 Variable declarations and assignments. Typeling features mutable, statically-typed variables. Variable declarations in Typeling require a name and a type or an initial value. Variable names must match the following regular expression:

```
[_a-z][_a-zA-Z0-9]*
```

As a result, identifiers which start with a capital letter cannot be used as variable names, they are instead reserved for type declarations.

A complete variable declaration will look like this:

```
x : i64 = 10;
```

However, if an initial value is provided, the type signature can be left out, as the compiler will infer it:

```
x := 10; // x is i64
```

Otherwise, the type of an uninitialized variable must be specified, as such:

```
x : i64;
```

Once defined, variables can be reassigned to new values with the "`=`" operator:

```
x := 10;   // x = 10
x = x + 1; // x = 11
```

4.4.3 Blocks. In Typeling, blocks can be used anywhere inside a function's body to define a new scope. This allows variable shadowing, for example:

```
fn one() -> i64 {
    x := 1;     // x = 1
    {
        x := 2; // x = 2
    }
    return x;   // x = 1
}
```

here, `x` is first declared with value `1`. Then, a new scope is created using the block statement, which allows `x` to be shadowed with value `2`. After the block statement ends, the top-most scope is removed and the value of `x` is `1` again.

4.4.4 Conditional branching. Conditional branching is achieved in Typeling through the use of `if` statements, which have the following structure:

```
if condition { /*then block*/ }
else { /*else block*/ }
```

The *condition* must be an `i64` value. If the condition is `0`, the control flow will go to the `else block`, otherwise it will go into the `then block`. Then, execution will continue with the next statement after the `if`. The `else block` is optional, therefore the following is a valid `if` statement:

```
if condition { /* then block */ }
```

in which case the control flow will only go into the `then block` if the *condition* is not `0`. Then it will continue with the next statement.

4.4.5 Loops. The `while` statement is Typeling's way of expressing loops. Just like *condition* and a *block*. When the program encounters a `while` statement, it first checks whether the *condition* is any value other than `0`. If that is the case, the control flow is moved inside the while block, otherwise, it continues onward. This process is repeated every time the program reaches the end of the *while block*. For example, an infinite loop can be defined as follows:

```
while 1 { /* while block */ }
```

4.4.6 free statement. In its current state, the Typeling prototype has to rely on manual memory management. Therefore, it makes sense for it to have a `free` statement, which acts like a function call. The `free` statement can be called on a user-defined data structure to deallocate its memory, as such:

```
type Num(i64)
fn main() {
    x := Num(10);
    free(x);
}
```

4.5 Expressions

Unlike statements, Typeling expressions are instructions which must produce some value. Currently, the following expressions are supported.

4.5.1 Integer operations. Typeling has support for addition (`+`), subtraction (`-`), multiplication (`*`), integer division (`/`) and the remainder (`%`) operators, which can be used on the `i64` type.

```
a := 1 + 2; // a =  3
b := 1 - 2; // b = -1
c := 1 * 2; // c =  2
d := 1 / 2; // d =  0
e := 1 % 2; // e =  1
```

4.5.2 Boolean operations. There are no boolean primitive types in Typeling. Instead, `i64` values are used, where `0` evaluates to `false`, and anything else to `true`. As a result, Typeling supports the boolean operators "`and`", "`or`", and "`not`":

```
a := -1 and 0;  // a =  0
b :=  3 or  0;  // b = -1
c := not 0;     // c = -1
d := not (-1);  // d =  0
e := not 2;     // e =  0
```

4.5.3 Function calls. Naturally, functions with a return type are also expressions, since calling them would produce a value. Other expressions can be passed as arguments to the function call, by writing them inside the parenthesis, separated by commas.

```
x := one();
three := sum(1,2);
```

Functions can also be used as statements, which will discard their result. Procedures, or functions which return the `()` unit type can only be called as statements, otherwise, they would produce a compilation error. For example, the `printf` C library function can be called like this:

```
printf("Hello world!\n");
```

4.5.4 Constructing data structures. Similarly to functions, data constructors can also be called to produce a new instance of their data type. For example, a `List` which contains the elements `1` and `2` can be constructed as such:

```
list := Cons(1, Cons(2, Nil));
```

4.5.5 Pattern matching. To extract the values from user-defined data structures, one would have to use pattern matching. In Typeling, pattern matching is achieved using the `case` expression. This expression consists of a series of pattern branches and their respective return values. Each branch in a case expression is evaluated top-to-bottom, and the value of the first match is returned. The following base patterns are supported: *data constructors* (e.g., `Tuple(/*...*/)`), *integer values* (e.g., `5`), *named wildcards* (e.g., `x`), and the *anonymous wildcard* (`_`). The base patterns can be nested to form more complex ones. For example, the **tail** of a `List` can be extracted using the following `case` expression:

```
tail := case list {
    Cons(_, xs) => xs,
    _ => Nil,
};
```

Pattern-matching expressions can also be applied to integers, through comparison against their values. For example, the `case` expression can be used as a ternary conditional operator, such as the following example, which evaluates the `condition`, and returns `100` if it is *false* (`0`) or `-20` otherwise:

```
x := case condition {
    0 => 100,
    _ => -20,
};
```

4.6 Implementation notes

The Typeling system consists of a just-in-time compiler, written in Rust [7] using LLVM API as its back-end. We preferred a dynamic compilation approach because it could potentially greatly improve the performance of the programs [12]. The source code of the compiler has been published on GitHub [13].

We chose to implement ADTs using a tagged union approach. As a result, the LLVM types for each of the data structure's constructors are generated first. Then, for the data type itself, we create an LLVM structure with two fields: an `i64` field for the tag of the constructor and a union of its constructor types as the data field. Figure 6 illustrates the resulting LLVM intermediate representation for the `List` data structure defined in Figure 5:

```
%List = type { i64, %constructor_Cons }
%constructor_Cons = type { i64, %List* }
%constructor_Nil = type {}
```

Fig. 6. Generated LLVM IR for the List type

5 EXAMPLES

Figures 7, 8, 9 and give a complete example of a Typeling program. It illustrates how the Typeling language can be used to create new data structures, as well as functions that operate on them. Specifically, the program presents the declaration and functions of a binary tree (Figure 7) and a linked list (Figure 8). Lastly, in Figure 9, the **tree sort** algorithm is implemented and the entry point of the program is defined, which uses this algorithm to sort an unordered list.

5.1 Ordered binary trees

Figure 7 contains the implementation of an ordered binary tree, the `BinTree` type. A binary tree is defined to be either a `Leaf`, which is empty, or a `Branch` containing an `i64` value and two `BinTree` children. To operate on binary trees, the following functions have been defined:

- `insert` - adds a new number to the binary tree, ensuring it remains ordered.
- `to_list` - does an inorder tree traversal and returns a linked list with its elements.
- `free_tree` - recursively deallocates the memory of a tree.

Note that the append function, which is called by `to_list` is defined in Figure 8.

```
type BinTree = Branch i64 BinTree BinTree | Leaf

fn insert(x: i64, root: BinTree) -> BinTree {
  return case root {
    Branch (y, left, right) => case x < y {
      0 => Branch (y, left, insert(x, right)),
      _ => Branch (y, insert(x, left), right),
    },
    _ => Branch (x, Leaf, Leaf),
  };
}

fn to_list(root: BinTree) -> List {
  return case root {
    Branch (x, left, right) =>
      append(
        to_list(left),
        Node (x, to_list(right))
      ),
    _ => Empty,
  };
}

// recursively frees a tree
fn free_tree(root: BinTree) -> i64 { /* ... */ }
```

Fig. 7. Example Typeling program: Binary Trees

```
type List = Node i64 List | Empty

fn append(xs: List, ys: List) -> List {
  return case xs {
    Node (x, xs) => Node (x, append(xs, ys)),
    _ => ys,
  };
}

fn to_tree(list: List) -> BinTree {
  return case list {
    Node (x, xs) => insert(x, to_tree(xs)),
    _ => Leaf,
  };
}

fn free_list(list: List) {
  while case list { Empty => 0, _ => 1} {
    tail := case list {
      Node(_, xs) => xs,
      _ => Empty,
    };
    free(list);
    list = tail;
  }
  free(list);
}
```

Fig. 8. Example Typeling program: Linked List

5.2 Linked list

The implementation of a linked list is illustrated in Figure 8. It is defined as either being an `Empty` list, or a `Node` containing an integer and another child `List`. Likewise, the following functions have been defined:

- `append` - appends the contents of a list (`ys`) to the end of another (`xs`).
- `to_tree` - creates a new ordered binary tree, containing all elements from a list.
- `free_list` - iteratively frees a list.

Unlike all the previously defined functions, `free_list` is written in an imperative style. Instead of recursion, it uses a `while` loop to iterate through each element in a list. The first `case` expression uses pattern matching to check whether the `list` local variable is not `Empty`. Then, it destructures the `tail` of the current element through the use of another `case` expression. Lastly, it deallocates the current list node and sets the `list` variable to its tail.

5.3 Tree sort

The definition of the `treesort` function can be seen in Figure 9. Its implementation converts an unordered list to an ordered binary tree with `to_tree`, then flattens the resulting tree with the `to_list` function, returning a sorted list. Figure 9 illustrates an example entry point to a Typeling program. The `main` function creates an unordered `list` and sorts it using the **tree sort** algorithm two times: first by manually calling the `to_tree` and `to_list` function, then by calling the `treesort` function.

6 DISCUSSION

The Typeling language closely adheres to the requirements provided in Section 2, given that we have designed it according to them. It successfully manages to demonstrate several strengths of ADTs in an imperative programming paradigm. It features HOPE-style *data constructors*, which facilitate the easy creation of user-defined data types. The `case` expression serves both as a control flow construct and a way of accessing the data of ADTs via pattern-matching.

The Typeling compiler implements ADTs as tagged unions, whose tag field represents the constructor used to initialize the data structure and the data field is a union of the type's constructors. We chose to implement a just-in-time compiler for Typeling, with the goal of improving its performance. However, Typeling programs will benefit from dynamic compilers only if the time to compile at run-time is lesser than the execution time savings of the optimizations [9]. Therefore, a static approach to the compilation process should also be considered.

As an imperative language, Typeling allows for precise control over the execution of a program. For example, it allows using `while` statements to implement loops and iterate over a data structure, instead of relying on recursive functions. Because of sequential

```
fn treesort(list: List) -> List {
  return to_list(to_tree(list));
}

fn main() {
  list := Node(10, Node(4, Node(8, Node(3, Node(6, Node(5, Node(7, Node(9, Node(1, Node(2, Empty)))))))))));
  tree := to_tree(list);
  print_list(list);
  print_list(to_list(tree));
  print_list(treesort(list));
  free_tree(tree);
  free_list(list);
}
```

Fig. 9. Example Typeling program: main function

composition, its syntax is easier to learn as well, given that it reads as a sequence of instructions. However, it is much harder to construct proofs to verify Typeling programs, compared to declarative ones, because of its use of mutable state, imperatives and jumps [15].

With that being said, Typeling supports a more functional approach to the development of its programs as well. This is because the language facilitates the creation of pure functions, due to them being implemented as call-by-value. In fact, in its current state, without the use of recursive function calls, some problems are challenging to solve or perhaps impossible. This is a result of the data structures being immutable. While it is possible to read the data of an ADT through pattern-matching using the `case` expression, there is no way of writing to it. Instead, Typeling programs rely on creating new instances of the data structure for every modification. As a result, current Typeling programs are prone to leaking memory, since its management is left up to the user. Hence, the following features are deemed necessary to improve the expressiveness and memory safety of the language.

6.1 Member access

To allow for the mutability of data structures, member access would have to be implemented. For example, in the case of a `Tuple` or `Point2D` type, whose definitions are illustrated below:

```
type Tuple (i64, i64)
type Point2D { x: i64, y: i64 }
```

mutating their data requires a new instance to be created, and the previous one to be deallocated:

```
t := Tuple(1, 2);
temp := t;
t = case t {
    Tuple(_, y) => Tuple(10, y + 10),
    _ => Tuple(0,0),
}; // t = Tuple(10, 12)
free(temp);
```

On the other hand, if the language supported accessing the members of the `Tuple` directly, both the `case` expression and the `free` statement could be omitted:

```
t := Tuple(1, 2); // t = Tuple(1, 2)
t.0 = 10;         // t = Tuple(10, 2)
t.1 = t.1 + t.0;  // t = Tuple(10, 12)
```

In addition, meaningful names could then be assigned to data constructors' fields, which would increase the clarity of the program, as is the case with the `Point2D` data structure:

```
p := Point2D(0, 0); // Point2D(0, 0)
p.x = 3;            // Point2D(3, 0)
p.y = 2;            // Point2D(3, 2)
```

6.2 Destructuring assignments

Destructuring assignments would allow the language to support pattern matching more imperatively and succinctly. For example, consider the previously defined `Tuple` type. To extract the first parameter, one must use a `case` expression, which can get quite verbose:

```
x := case Tuple (10, 20) {
    Tuple (y, _) = y;
    _ => 0;
};
// x = 10
```

This results from having to write the entire case expression, which always requires a default case. However, data structures with a single constructor will always have a single possible case, so the default branch is unreachable. Therefore, the Typeling language could be improved by implementing pattern-matching assignments for types with a single *data constructor*:

```
Tuple (x, _) := Tuple (10, 20);
// x = 10
```

6.3 Garbage collection

We consider garbage collection to be a required feature of Typeling. Because our prototype relies on manual memory management, the compiler cannot guarantee the program's memory safety. A novice programmer may struggle with this method of memory management, and their programs might be at risk of having memory leaks. Other memory management models, such as Rust's borrow checker,

can guarantee memory safety without sacrificing performance, but they have a steep learning curve as well. Implementing garbage collection in Typeling would benefit beginners the most, allowing them to write code without worrying about memory management.

7 FURTHER WORK

The Typeling programming language is still in its early stages, thus there is still a lot of room for improvement left. We consider the following features to be appropriate next steps in any further design of the language:

7.1 Polymorphic types

Polymorphism entails defining procedures that can be applied to a wide range of objects [18]. Polymorphic data types, specifically parametric polymorphism, would greatly benefit the Typeling language, allowing users to model even more complex data structures and allow code reuse. Some examples of languages which support parametric polymorphism are Haskell [17] and Rust [7].

7.2 Generalized algebraic data types

Generalized Algebraic Data Types (GADTs) are a generalization of algebraic data types (ADTs), that allow more precise type representations and pattern matching [21]. GADTs can define custom data types with refined type constraints, enabling stricter type checking and improved type safety. Furthermore, GADTs can enhance code readability by providing a more precise representation of the data in the type system. For example, GADTs would make it possible to write a safe evaluator for simply-typed object languages, which does not require values to carry run-time tags [21].

7.3 Optimizing memory usage

Memory usage optimization is crucial for improving the performance of a programming language. Currently, the Typeling compiler allocates memory on the heap for every new data constructed, even for *unit* types, which are data constructors that do not have any parameters. *Unit* data structures, on the other hand, are immutable because their only data is the constructor's constant tag. As a result, one potential improvement would be to allocate this type of data structure once and then share references to the same memory address among all instances.

7.4 Better type inference

Type inference is an important feature that allows the compiler to automatically deduce the types of variables and expressions in a program. Implementing more advanced type inference in Typeling would allow us to simplify the syntax even further. For example, the compiler could infer the type declarations in function signatures, allowing users to omit them.

ACKNOWLEDGMENTS

I want to thank my research project supervisor, Peter Lammich, for his support throughout this project. I am deeply grateful for the time he invested in our regular meetings and his valuable feedback, which helped me shape the design of my language.

Furthermore, I would like to extend my gratitude to my family and friends, whose support has been a constant source of motivation.

REFERENCES

[1] 2021. The Java® Language Specification. https://docs.oracle.com/javase/specs/jls/se16/html/index.html
[2] 2023. DataEnum. https://github.com/spotify/dataenum original-date: 2017-12-14T14:55:41Z.
[3] 2023. ECMAScript® 2024 Language Specification. https://tc39.es/ecma262/
[4] 2023. OCaml. https://github.com/ocaml/ocaml original-date: 2012-11-20T22:18:22Z.
[5] 2023. The Python Language Reference. https://docs.python.org/3/reference/index.html
[6] 2023. The Rust Programming Language. https://github.com/rust-lang/rust original-date: 2010-06-16T20:39:03Z.
[7] 2023. The Rust Reference. https://doc.rust-lang.org/stable/reference/
[8] 2023. The TypeScript Handbook. https://www.typescriptlang.org/docs/handbook/intro.html
[9] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. 1996. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation (PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 149–159. https://doi.org/10.1145/231379.231409
[10] Brandt Bucher and Guido van Rossum. 2020. PEP 634 – Structural Pattern Matching: Specification. https://peps.python.org/pep-0634/
[11] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. 1980. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming (LFP '80)*. Association for Computing Machinery, New York, NY, USA, 136–143. https://doi.org/10.1145/800087.802799
[12] Craig Chambers. 2002. Staged compilation. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation (PEPM '02)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/503032.503045
[13] Victor Hornet. 2023. Typeling. https://github.com/victorhornet/typeling original-date: 2023-05-10T21:38:14Z.
[14] ISO/IEC. 2020. ISO International Standard ISO/IEC 14882:2020(E) – Programming Language C++. https://isocpp.org/std/the-standard
[15] P. J. Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (March 1966), 157–166. https://doi.org/10.1145/365230.365257
[16] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, Kc Sivaramakrishnan, and Jérôme Vouillon. 2022. *The OCaml system release 5.0: Documentation and user's manual*. report. Inria. https://inria.hal.science/hal-00930213 Pages: 1.
[17] S. Marlow. 2010. Haskell 2010 Language Report. https://www.haskell.org/onlinereport/haskell2010/
[18] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4
[19] Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. 1997. *The Definition of Standard ML*. The MIT Press. https://doi.org/10.7551/mitpress/2319.001.0001
[20] Stack Overflow. 2023. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/#most-popular-technologies-language
[21] François Pottier and Yann Régis-Gianas. 2006. Stratified type inference for generalized algebraic data types. *ACM SIGPLAN Notices* 41, 1 (Jan. 2006), 232–244. https://doi.org/10.1145/1111320.1111058