Position Papers of the 18th Conference on Computer Science and Intelligence Systems pp. 67–76
DOI: 10.15439/2023F5882
ISSN 2300-5963 ACSIS, Vol. 36

Decoupling Types and Representations of Values for Runtime Optimizations*

Krajča Petr† and Škrabal Radomír
0000-0003-4278-3130
0000-0003-3929-7238
Palacky University Olomouc
17. listopadu 12, CZ77146 Olomouc
Czech Republic
E-mail petr.krajca@upol.cz, radomir.skrabal01@upol.cz

***Abstract*—The way data is stored in a computer's memory is crucial from the performance point of view. The choice of the most appropriate data structure depends not only on the algorithm but also on the input data itself. Unfortunately, this information may not always be known in advance, requiring programmers to make educated guesses about the data's characteristics. If these guesses are inaccurate, it can result in suboptimal performance. To address this challenge, we introduce a novel programming language that draws inspiration from database management systems. This language has the capability to automatically select optimal data structures and, consequently, algorithms based on the input data to improve performance.**

I. Introduction

DATA types play an increasingly important role in modern programming. In fact, it is not a single role, but four roles at once. Data types (i) create abstractions, (ii) define variable domains (which values can variable hold), (iii) define representation of values (how data are stored in memory), and corollary (iv) assists to dispatch operations with data [1].

Much focus has been given to the role (ii) of data types which allows to build a more robust software, hence simplify software development. We are to explore the role (iii) which deals with value representation, since it has significant impact on performance, due to the role (iv) of the type system.

Let us recall what is meant by type, value, and its representation. For simplicity we are to assume that *a type* is a set of values. *A value* can be seen as an abstract entity which has no location in time or space. However, a value may have encoding that can be represented in the memory of the computer. This encoding of a value shall be called *representation* and can be assigned to a variable. Naturally, a single value may have multiple representations. For instance *value 42* may be represented as "42" (two numerals), "forty two" (words), "XLII" (roman numerals), 00101010 (binary number), etc. All of these are valid representations of the number, however, each is suitable for different situations. Note that the notions of a *type* and *representation* are orthogonal. The type of a variable provides information on values that can be assigned to a given variable, while representation is an encoding of a value, see [2]. Types play an important role during the compilation process, since they allow to discover invalid operations, representations are crucial for program execution since they determine how the program is to be executed. For example, points can be represented in Cartesian or polar coordinates while each representation is suitable for different applications. Analogously, in data analysis various representations of vectors (dense or sparse) and matrices (dense, sparse, row/column major) are used. Which of these representations is more appropriate is determined by the algorithm and often by the input data itself.

In contemporary programming languages decoupling of types and their representations is achieved via interface inheritance (Java, C#) or type-classes (Haskell). These time-approved techniques have a particular downside. The selection of a type representation is often made in compile-time based on more-or-less accurate assumptions on the future use of the program. If the data representation does not fit the actual input data, it may lead to a suboptimal performance, e.g., if a sparse matrix is stored as a dense matrix in memory.

To deal with this kind of issues we propose a programming language where types and representations of values are strictly decoupled. Our motivation comes from the world of relational database management systems (RDBMS). For instance, RDBMS stores tuples in a sequential data file. Depending on the size of data, meta-data and query, RDBMS during query processing selects the most suitable representation of the data from the data file. Let us consider join of two data tables in RDBMS. If the data are small enough then the input is read and processed sequentially using some trivial algorithm (e.g. nested loop join). If the data are larger, RBDMS can change their representation to process it faster. For instance, to a hash table and process data using a hash-join. Or, if an index is available, data can be turned into an ordered set and processed using a sort-merge-join algorithm, see e.g. [3] for details.

Our intention is to design a language where this capability, i.e., select the most appropriate data representation and by extension most appropriate algorithms, is available in a general purpose programming language. The intended applications come from the area of data analysis where the selection of an appropriate data representation is of a crucial importance.

*Supported by grants IGA_PrF_2022_018 and IGA_PrF_2023_026 of Palacký University Olomouc.
†Corresponding author.

©2023, PTI

The paper is organized as follows. Section II provides introduction to the basic of the language we propose. Then, Section III describes the concept of an *extended function* allowing to dispatch functions based on types and values of arguments. Sections IV and V are devoted to the type inference in our language. Paper concludes with Sections VI and VII on the experimental evaluation and notes on the current state of the language in the future.

II. Language Velka

We propose a programming language called Velka as an experimental framework, where types and representations are strictly separated from each other. Velka is loosely based on Scheme [4] and has a Lisp-like syntax.

Velka is strongly typed, detecting many type-related errors at compilation time, with no automatic type coercion. However, Velka can convert between multiple representations of a single type if it is viable. The evaluation method (strict or lazy) for Velka is not specified.

The following subsections are brief introduction to Velka language. We tacitly assume that readers are acquitted with the Lisp-like language [4], [5], [6], therefore we focus on the most essential parts of the language.

A. Type and representation signatures

Representation signature is a way of expressing a representation statically in Velka syntax. Atomic types use their name as a signature like `Int`, `Bool` for integers or Boolean values. An atomic representations uses colon `:` to separate the type and its representation, for example `Int:Native` and `Int:Roman` denote two different representations of integers, (i) a binary representation in the memory and (ii) a string of Roman numerals. Type signatures of tuples are written in parentheses, e.g., `(Int Bool)`.

B. Expressions

Velka follows in the majority of cases syntax and semantics of the Lisp-family languages. Expressions are either atoms (literals, symbols) that evaluate to their associated values or lists, e.g., `(+ 1 2)` or `(+ 1 (* 2 3))`. Lists are evaluated recursively as follows. First, the first element of the list is evaluated. If it evaluates to a function, then all remaining elements (i.e., arguments) are evaluated, and the function is applied on these values. Besides functions, the first element may evaluate to a special form (`if`, `define`) that is a special operator that controls evaluation of its operands.

C. Simple functions

Functions in Velka are created with the special form `lambda`. It creates a function and returns it as a value. The special form `lambda` accepts two arguments: (i) a list of formal arguments and (ii) an arbitrary expression, called body of the function. For example, if we are to create a function which adds two values, we use the following code:

```
(lambda (x y) (+ x y))
```

If a particular type or representation is required. We can specify the type and its representations of the formal arguments of a function. As shows the following example:

```
(lambda ((Int:Native x) (Int:Native y))
  (+ x y))
```

D. Defining types and representations

Definition of a type in Velka is done using the special form `type`. It accepts one argument—a symbol with a name of the type. For example, a set of integers is defined as follows.

```
(type IntSet)
```

Note that type is merely a declaration of a type name. Its semantics is defined further in the program through its representation.

Representations are defined with the special form `representation`. It accepts two arguments—a name of the representation and the name of an existing type. To create representations of a set of integers based on linked lists and bit-vectors one could use.

```
(representation LinkedList IntSet)
(representation BitVector IntSet)
```

Just like the `type` special form, the `representation` special form creates only a declaration. An actual representation of a value is given in a *constructor*, a special kind of function that creates a value and marks it with the representation it is constructing.

To define a constructor in Velka, we use the `constructor` special form. Its first argument is a signature of an atomic representation, the second argument is a list of formal arguments of the constructor, and the last argument is an expression, which returns the constructed value. The value is created with the `construct` special form. For example, the constructor is created and used as follows:[1]

```
(constructor IntSet:BitVector () 0)
(construct IntSet BitVector)
   ;;<0 IntSet:BitVector>
```

E. Deconstruction and Conversions

Naturally, it is necessary to extract a partial information from a value. For this purpose Velka uses the *deconstructors*. The special form `deconstruct` extracts the original value from a value created by a constructor. It accepts two arguments—(i) the deconstructed value and (ii) a representation signature. When evaluated the `deconstruct` special form tries to unpack the deconstructed value into a value with the representation specified as the second argument. If the deconstructed value cannot be unpacked into such value, the execution ends with an error. For example, suppose we want to check if a set, represented by a previously defined `IntSet:BitVector` contains a number 3. We extract the integer value representing the set and then use the bitwise AND operation to check if the third bit is set.

[1]This is a simplified example capable that covers only the values from 0 to 31 for brevity. In practice, a more elaborate solution would be possible and necessary.

```
(define empty-set (construct IntSet BitVector))
(equalp 0
  (bit-and 8
    (deconstruct empty-set Int:Native))) ;;#f
```

Conversions between representations are declared with the special form `conversion`. It accepts four arguments (i) a representation from which a value is converted, (ii) a representation to which a value is converted, (iii) name of an argument, and (iv) body of a conversion. For example:

```
(conversion IntSet:BitVector IntSet:LinkedList
  (IntSet:BitVector set)
  (...))
```

Conversions are performed either implicitly (in most cases) or explicitly by the special form `convert`.

F. let-type and type variables

Velka automatically considers all the symbols in type signatures to be type atoms. To use type variables in type and representation signatures a `let-type` special form must be used. It accepts two arguments—(i) a list of type variable names and (ii) an expression. It allows us to use type variables in the expressions. For example, if we want to make a function that adds an arbitrary value into a collection, we may use the following code:

```
(representation Demonstration List)
(define my-add (let-type (A)
   (lambda ((List:Demonstration l) (A element))
     (tuple element l))))
```

G. Built-in representations and operators

For convenience Velka contains several built-in representations that come from the host environment, in our case from the Java platform. This covers the native representations of primitive data types (integers, floating point numbers, string), lists (ArrayList, LinkedList), tuples, or sets.

Naturally, Velka contains also built-in operators to manipulate these built-in representations. We have already stumbled upon some of these like: +, `bit-and`, `tuple`, or `equalp`. It is out of scope of this text to give a full list of these operators. Thus, we explain those used in this text as we go. For the full list of built in operators, please refer to the documentation [7].

III. Extended functions

The key feature of Velka are *extended functions*. The purpose of the extended functions is to bring capabilities of the RDBMS to a general purpose language, allowing Velka to select the most appropriate data representation and algorithm based on the input data. This allows to optimize the program execution without an explicit involvement of a programmer

A. Multiple implementations

Broadly speaking an *extended function* is a function that can select the code to run, depending on the arguments it is applied to. For example, suppose we have an extended function `intersect` that performs an intersection of two sets. We have two representations of sets: the `Set:LinkedList` and the `Set:BitVector` represented by linked lists and bit vectors, respectively.

We can apply `intersect` with either both arguments being a `Set:LinkedList`, both being a `Set:BitVector` or one of each. All of these applications are correct, because the arguments are of the type `Set`. However, they can be of a different representations.

The extended function `intersect` decides during the application which algorithm to use. When both arguments are `Set:BitVector` values, it chooses binary operations. If one argument is a `Set:BitVector` and second is a `Set:LinkedList`, it converts the second argument. If both arguments are `Set:LinkedList`, it uses merge algorithm. Or if the extended function considers it more efficient, it can convert both arguments into the `Set:BitVector`, and use the bit-wise AND operation.

From the technical point of view an *extended functions* can be considered a pair $\langle I, cost \rangle$, where I is a set of simple functions $\{f_1, f_2, \ldots, f_n\}$ called *implementations* and *cost* is a function that associates each implementation with a cost function. Implementations are algorithms that the extended function chooses from.

All implementations must have the same type; however, they must have a different representation than the other implementations. This ensures that each implementation accepts the same number and types of arguments, and returns a value of the same type. Therefore it can be applied to the arguments of the extended function.

A cost functions $cost(f_i)$ associated with each implementation are responsible for the selection of an implementation applied by extended functions.

B. Running Example

We use type `Name` describing names of persons as a running example in the following text. Depending on the application, in some cases it may be reasonable to represent a given and a family name separately, and in some cases represent them as a single name. Hence, it makes sense to consider a representation `Strd` (structured) consisting of pair of strings, and a representation `Ustrd` (unstructured), consisting of a single string.

In Velka such type and representations are implemented by the following code:

```
(type Name)
(representation Strd Name)
(constructor Name Strd
  ( (String:Native firstname)
    (String:Native surname))
  (tuple firstname surname))
(representation Ustrd Name)
(constructor Name Ustrd
  ((String:Native name))
  name)
```

We assume the following functions for extracting underlying `String:Native` values:

```
(define get-name (lambda ((Name:Ustrd name))
```

```
  (deconstruct name String:Native)))
(define get-given-name
  (lambda ((Name:Strd name))
    (get (deconstruct name
           (String:Native String:Native))
         0)))
(define get-family-name ...) ;; analogously
```

C. Extended functions in Velka

We use the special form `extended-lambda` to create an extended function. It accepts a single argument—a list of type signatures of its arguments. Initially, an extended function does not contain any implementations. For example, consider an extended function `name-equalp` testing equality of names. We start with a definition of an extended function with two `Name` arguments:

```
(define name-equalp
  (extended-lambda (Name Name)))
```

To add a new implementation into an existing extended function, we use the special form `extend`. It accepts three arguments—(i) an extended function, (ii) a simple function, and (iii) a cost function. The cost function, however, is not mandatory. We discuss cost functions in more detail in Section III-F.

For example, if we want to add an implementation for `Ustrd` representation, we use the following code. Also please note, that the `extend` special form does not cause side-effects. It creates a new extended function from the argument, adds a new implementation to it, and returns it. The original extended function is not affected. It is therefore necessary to rebind the symbol `name-equalp` with the special form `define`.

```
(define name-equalp
  (extend name-equalp
    (lambda ((Name:Ustrd first)
             (Name:Ustrd second))
      (equalp (get-name first)
        (get-name second)))))
```

Not any simple function can become an implementation. A simple function must accept the same number and the same type (but not the representations) of arguments as the extended function. Also, the first added implementation sets the return type of the extended function. Therefore any implementation after that must return a value of the same type (but not necessarily of the same representation).

D. Representation of extended function

We cannot use the applicable representation directly to describe a representation of an extended function. Since the implementations can have different representations of arguments and return values.

To resolve these ambiguities we introduce a concept of a *representation set*. It is a finite set of representations, where each representation belongs to the same type. The representation of an extended function is a representation set, containing representation of each implementation.

For example, function `name-equalp` clearly has the type $[\texttt{Name,Name}] \rightarrow \texttt{Name}$. However, `name-equalp` contains implementation with the `Name:Strd` and the `Name:Ustrd` representation of the argument. Therefore, its representation is $\{[\texttt{Name:Strd,Name:Strd}] \rightarrow \texttt{Bool:Native}, [\texttt{Name:Ustrd,Name:Ustrd}] \rightarrow \texttt{Bool:Native}\}$.

E. Application of extended function

Let's say we apply the extended function $\langle I = \{i_1, i_2, \ldots\}, cost\rangle$ on the arguments $a_1, \ldots, a_n$. Each argument a_j is evaluated to the value e_j. Then, the extended function iterates through each implementation i_k. Each implementation i_k has the cost function $cost(i_k)$.

The cost function takes the same type of the arguments as the extended function and returns an integer. Therefore, we can compute cost for each implementation with respect to the arguments, i.e., $(cost(i_k))(a_1, a_2, \ldots)$.

The extended function searches for an implementation i such that, its cost is minimal with respect to the arguments. Then the implementation i is applied as a simple function. For example, let's say we apply our function `name-equalp` with some arguments of `Name:Ustrd` representation.

```
(name-equalp
   (construct Name Ustrd "John Doe")
   (construct Name Ustrd "James Dee"))
```

First both arguments are evaluated, getting values `"John Doe"` and `"James Dee"` both of the `Name:Ustrd` representation.

The exrended function `name-equalp` has two implementations—(i) accepting two arguments of the `Name:Ustrd` representations and (ii) accepting two arguments of the `Name:Strd` representation. We shall call these representations the unstructured implementation and the structured implementation respectively.

Both implementations are defined without a cost function. Therefore, the default cost function is used. The default cost function returns the number of arguments that are not in the expected representation for the implementation.

In our example, the unstructured implementation is expecting two arguments of the `Name:Unstructured` representation. Both arguments have the `Name:Unstructured` representation; therefore default cost function yields 0. On the other hand the structured implementation is expecting two arguments of the `Name:Structured` representation. Since both arguments have the `Name:Unstructured` representation, the cost function yields 2.

The unstructured implementation has the least cost and is applied on the arguments. If there are two or more implementation with the lowest cost, we do not specify which one is selected.

F. Cost functions

A cost function is a function (extended or simple) that is associated with each implementation of an extended function. It computes cost of an implementation for the given arguments. Formally speaking, for the implementation i with the argument

types $t_1, t_2, \ldots, t_n$, a cost function can be any function with the type $(t_1, t_2, \ldots, t_n) \rightarrow$ Int.

Let's take the unstructured representation from the previous section. For this implementation a cost function must have the type (Name Name) $\rightarrow$ Int.

Let's say this implementation is superior in performance, and we want to use it, even if only one of the arguments has the Name:Ustrd representation. We use the following cost function:

```
(lambda ((Name:* first) (Name:* second))
  (if (or (instance-of-representation first
             Name:Ustrd)
          (instance-of-representation second
             Name:Ustrd))
    -1 3))
```

This cost function takes two arguments of the Name type. Notice that we do not specify the representation of the cost function arguments. Since internally a cost function is applied like any other function, enforcing a specific representation would cause conversion of the representation during application. Therefore special form instance-of-representation would not work as expected.

When we apply this cost function with at least one Name:Ustrd argument, the function yields -1, which is less than the default cost function yields (the default cost functions minimum is 0). On the other hand, if neither argument is Name:Ustrd, the cost function yields 3, which is above what default cost function can yield. Therefore, with this cost function the extended function name-equalp selects the unstructured implementation, if at least one of the arguments is Name:Ustrd.

To maximize flexibility, we impose minimal restrictions on cost functions. However, this approach introduces challenges as evaluations may fall into non-trivial infinite loops or lead to severe performance loss. User must be aware of these caveats and work accordingly.

IV. Type Inference and Unification

Velka is a strongly typed functional programming language. The type inference and unification are crucial parts of the language design. Especially, since Velka distincts between types and their representations. In this section we outline the key and the most distinctive parts of the type system. Detailed description is to be discussed in the extended version of the paper.

A. Type and representation unification

Our type unification algorithm is a combination of J. W. Lloyd's [8] and Hindley-Milner's [9] approach. It is explicitly returning either an unification substitution or a false answer, while maintaining a predictable and deterministic traversal through a type structure avoiding any side effects during its computation.

The unification of representations is in principle the same as the unification of types. We extended the type unification algorithm for the representation in a straightforward manner. However, we do not introduce representation variables, we use only type variables, which are handled the same way as in type unification. The *representation set* is unique for the representation.

A representation set is meant to represent a set of possible representations of the value. We can unify a representation set with an other representation, if any representation in the set unifies with the other representation.

This trivially holds for a type variable, since it unifies with any representation substituted for it. For an atomic representation, the atomic representation must be present in the set and then they unify with the empty substitution. For functions and tuples, if set contains a representation that unifies over some substitution, the set unifies with the same substitution.

Only case of two representation sets remains. However, we can unify each representation of the first set with the second set. In a recursive call of the algorithm we unify each representation of the second set with the current representation of the first set. Thus we try to unify each representation of the first set with each representation of the second set.

B. Representation inference limitations

Since the representations can be interchanged freely one for another one of the same type (assuming there is a conversion), we cannot unambiguously infer the representation of an expression.

For example, let's consider the if expression. In case of the types, the inference rule for this special form is [1]:

$$\frac{c : \{true, false\}, t : A, f : A}{if(c, t, f) : A}$$

Meaning, if we have an expression c of the type $\{true, false\}$ and expressions t and f of the type A, we can infer the expression (if c t f) is of the type A.

On the other hand, let's take the Velka expression: (if (= a 0) 42 (construct Int Roman "XXI")). For types everything is clear, the first argument is a boolean and both the second and the third argument are integers. Therefore the expression yields an integer.

However, for representations, the expression yields either an Int:Native value or an Int:Roman value. At compile time we cannot decide, which integer representation the expression yields.

Therefore, the representation inference rule for the if expression is different:

$$\frac{c : \{true, false\}_{Native}, t : A_R, f : A_S}{if(c, t, f) : \{A_R, A_S\}}$$

This way we can unambiguously infer the representation of the expression from the previous example. Since (= a 0) has the representation $\{true, false\}_{Native}$, 42 has the representation Int:Native and (construct Int Roman "XXI") creates a value with the representation Int:Roman, the inferred representation of the expression is $\{\mathtt{Int:Native}, \mathtt{Int:Roman}\}$.

Similar ambiguity is linked with the semantics of extended functions. Therefore, the `extended-lambda` special form and applications of extended functions suffer the same predicament. These are discussed in more detail in Sections V-G and V-H.

V. Representation inference algorithm

An algorithm for representation inference in Velka is designed around the implementation of inference rules for expressions present in Velka. This includes the most frequent language constructs like function applications, tuples, literals, and lambda expressions.

An input of the algorithm consists of two values: e an expression whose representation is inferred and $\mathcal{E}$ an lexical environment, where the expression is inferred.

In this context, an environment is a map of symbols and their bindings. For the inference algorithm, it is important that a symbol is bound to an expression, that infers to the correct representation.

Argument $\mathcal{E}$ influences the inference of symbols as well as functions (since they carry their creation environment). Our algorithm creates a new environment during application inference in a similar manner a lexical closure does.

The inference algorithm returns a pair of values: a representation and a substitution. The substitution carries already inferred or partially inferred type variables over to the further computation. A description of inference rules for Velka expression types follows.

A. Literals and Construct special form

Literal expressions in Velka infers to their assigned representations. Returned substitution is always empty.

The special form `construct` consists of a constructed representation x_r and constructor arguments $a_1, a_2, \ldots, a_n$. The inferred representation is clearly x_r. We must also ensure, that constructor for the arguments exists, and that the arguments are of the correct types.

B. Symbols

When a representation of a symbol is inferred, we inspect if it has a binding in the environment hierarchy. If it has a binding to an expression e_b, we recursively call the inference algorithm on e_b. If a symbol does not have a binding, we cannot infer its representation, therefore we return a type variable and an empty substitution.

C. Substitution Merge

There is a class of type related errors, that occur when a single symbol is used twice, each time as a different type. We need a way to detect this incompatibility between substitutions, and also a convenient way to combine compatible substitutions into a single one. For this purpose we use the substitution merge algorithm.

The merge is based on the idea that two substitutions need to substitute the same variable in order for conflict to arise. If they do not have any common variable, we can use set union to merge them, getting a valid substitution. Note that, even if substitutions have a common variable A, the conflict only occurs if expressions on the right side of the substitution are not equal. Otherwise they are the same expression and set union produces a valid substitution.

Algorithm 1: Substitution merge algorithm

```
1 Function ∪_S(σ, φ)
2   while there is variable A such that A\e ∈ σ, A\f ∈ φ
      and e ≠ f do
3     ρ ←UnifyRepresentations(e, f);
4     if ρ = false then return false;
5     σ ← σρ ;
6     φ ← φρ;
7   return σ ∪ φ
```

Assuming we have two substitutions σ and ϕ, with a common variable A, such that $A\backslash e_\sigma \in \sigma$, $A\backslash e_\phi \in \phi$ and $e_\sigma \neq e_\phi$. If we can find an unifier ρ for e and f, we can compose it with σ and ϕ. In that case $\sigma\rho$ and $\phi\rho$ still have a common variable A, however $\rho(e) = \rho(f)$. Therefore $\sigma\rho \cup \phi\rho$ is a valid substitution.

This is the idea behind the Algorithm 1. We are iterating over common variables that are substituted for not equal expressions of σ and ϕ. For each such variable we find a unifier of the substituted expressions, and compose it with the sets. We end the loop when no such variable can be found and return a set union of the two substitutions. If at any time an unifier does not exists, the substitutions are conflicting and an error is thrown.

D. Special form `if`

We describe the inference of the `if` special form as an example on how a special form representation inference is handled in Velka. Other special forms and tuples are handled in a very similar manner.

The special form `if` takes form of `(if` c t f`)`, where c is the condition expression, t is a true branch expression and f is a false branch expression. The algorithm infers representations of each of these sub-expressions.

We make sure that the type of c is the boolean and that both t and f infers to the same type. Once the algorithm takes care of this basic type checking, it merges the substitutions of sub-expressions to the substitution for the `if` expression.

The inferred representation is a representation set of representations inferred for t and f, since at compile time it is not possible to decide which one is used. See Algorithm 2 for pseudo-code.

E. Lambda Expressions

A lambda expression assigns representations $r_1, r_2, \ldots, r_n$ to its formal arguments $a_1, a_2, \ldots, a_n$. We use a mock up environment and a special inference-only expressions called *representation holders* to reflect this.

A *representation holder* is a special expression that cannot be evaluated and it infers to an assigned representation with

Algorithm 2: Special form `if` inference algorithm

```
1  Function InferIfRepresentation(c, t, f, ℰ)
2      ⟨r_c, σ_c⟩ ← InferRepresentation(c, ℰ);
3      ⟨r_t, σ_t⟩ ← InferRepresentation(t, ℰ);
4      ⟨r_f, σ_l⟩ ← InferRepresentation(f, ℰ);
5      φ_c ← UnifyTypes(r_c, Bool)
6      if φ_c = false then raise error;
7      φ ← UnifyTypes(r_t, r_f);
8      if φ = false then raise error;
9      ψ ← σ_c;
10     ψ ← ψ∪_S σ_t;
11     if ψ = false then raise error;
12     ψ ← ψ∪_S σ_f;
13     if ψ = false then raise error;
14     return ⟨{r_t, r_f}, ψ⟩
```

Algorithm 3: Lambda expression inference algorithm

```
1  Function InferLambdaRepresentation([a_1, ..., a_n],
       [r_1, ... r_n], e_b, ℰ)
2      Γ ← ⟨{a_i ← e^{r_i} | i = 1, ... n}, ℰ⟩;
3      ⟨r_b, σ⟩ ← InferRepresentation(e_b, Γ);
4      return ⟨σ([r_1, r_2, ..., r_n]) → r_b, σ⟩
```

the empty substitution. We denote the representation holder's representation using an upper index; for example e^{x_r} or f^A.

When inferring a lambda expression, we create a new mock up environment Γ. Its parent is $\mathcal{E}$, the environment where the lambda expression is evaluated. Γ contains each formal argument a_i bound to the representation holder e^{r_i}. We use the environment Γ to infer the body e_b of the lambda expression. We denote the computed representation r_{body} and the used substitution σ.

It is tempting to use $[r_1, \ldots, r_n] \to r_{body}$ as the resulting representation. However, consider the following example:

```
(let-type (A) (lambda ((A a)) (+ a 1)))
```

In this case the assigned argument representation is A, r_{body} is `Int:Native`, and σ is $\{A\backslash\texttt{Int:Native}\}$. Therefore, $[r_1, \ldots, r_n] \to r_{body}$ is $[A] \to$ `Int:Native`. However, this lacks already known argument representation.

Thus we use $\sigma([r_1, \ldots, r_n]) \to r_{body}$ as a resulting representation to propagate information from the body inference. Used substitution is σ. You can see the pseudo code in the Algorithm 3.

F. Unifying representations on a type level

Velka allows to apply a function with correct types, but different than declared representations of arguments, assuming the conversions exists at inference time. Since we infer the representations of the expressions, we need a tool that ensures type safety, and allows different representations.

As discussed in IV-A, to unify types and representations we use the standard algorithm with minor modifications. The algorithm's pseudo-code is presented in Algorithm 4. It accepts representations instead of types and is able to unify representation sets on the type level.

Algorithm 4: Algorithm for type unification working with representations

```
1  Function UnifyRepresentationsAsTypes(s, t)
2      if s and t are representation atoms with the same type
         name or type variables with the same name then
3          return {}
4      else if s is a type variable then return {s \ t} ;
5      else if t is a type variable then return {t \ s} ;
6      else if s is a s_1 → s_2 and t is a t_1 → t_2 then
7          σ ← UnifyRepresentationsAsTypes(s_1, t_1);
8          if σ = false then return false;
9          φ ←UnifyRepresentationsAsTypes(σ(s_2), σ(t_2));
10         if φ = false then return false;
11         return σφ
12     else if s is a [s_1, ..., s_n] and t is a [t_1, ..., t_n] then
13         θ ← {};
14         for i ← 1, ... n do
15             ρ ← UnifyRepresentationsAsTypes
16                 (θ(s_i), θ(t_i));
17             if φ = false then return false;
18             θ ← θρ;
19         return θ;
20     else if s is a {s_1, s_2, ..., s_n} then
21         return UnifyRepresentationsAsTypes(s_1, t);
22     else if t is a {t_1, t_2, ..., t_n} then
23         return UnifyRepresentationsAsTypes(s, t_1);
24     else return false ;
```

G. Extended lambda and extend expressions

An extended lambda accepts arguments $a_1, a_2, \ldots, a_n$, with a user assigned types $t_1, t_2, \ldots, t_n$, respectively.

The special form creates an empty container, where implementations are added later. Therefore, we cannot infer a specific representation for the `extended-lambda` special form alone.

We use the representation sets in a form of $\{t_r | t_r$ is a representation of $t\}$, for a type t. This encompasses any possible representation of type t. For convenience we use the following notation: $t^* = \{t_r | t_r$ is a representation of $t\}$.

For an extended lambda (`extended-lambda` t_1 t_2 ... t_n) where $t_1, t_2, \ldots, t_n$ are argument types defined by the user, we infer $[t_1^*, t_2^*, \ldots, t_n^*] \to A$ where A is a new unused type variable. The substitution is empty.

For example, the `(extended-lambda (Int String))` expression infers to the pair $\langle[\texttt{Int}^* \ \texttt{String}^*] \to A, \emptyset\rangle$.

The other part of the extended functions is the special form `extend`. It has the form (`extend` e_{ext} e_{impl} e_{cost}), where e_{ext} evaluates into an extended function, e_{impl} evaluates into a simple function—the future implementation, and e_{cost} evaluates into the cost function. We put cost function aside for now.

Algorithm 5: Extend special form inference algorithm

1 **Function** INFEREXTENDREPRESENTATION(e_{ext}, e_{impl}, e_{cost}, $\mathcal{E}$)
2 $\langle r_{ext}, \sigma_{ext} \rangle \leftarrow$ INFERREPRESENTATION(e_{ext}, $\mathcal{E}$);
3 $\langle r_{imp_L} \to r_{imp_R}, \sigma_{impl} \rangle \leftarrow$ INFERREPRESENTATION(e_{impl}, $\mathcal{E}$);
4 **if** UNIFYREPRESENTATIONSASTYPES
5 (r_{ext}, $r_{imp_L} \to r_{imp_R}$) = false **then raise error**;
6 $\langle r_{cost}, \sigma_{cost} \rangle \leftarrow$ INFERREPRESENTATION(e_{cost}, $\mathcal{E}$);
7 **if** UNIFY-REPRESENTATION(r_{cost}, $\langle r_{imp_L} \to$ *Int:Native*) $=$ false **then raise error**;
8 **if** r_{ext} *has form of* $[r_1^*, r_2^*, \ldots, r_n^*] \to A$ **then**
9 **return** $\langle \{r_{imp_L} \to r_{imp_R}\}, \emptyset \rangle$
10 **return** $\langle e_{ext} \cup \{r_{imp_L} \to r_{imp_R}\}, \emptyset \rangle$

Representations r_{ext} and $r_{imp_L} \to r_{imp_R}$[2] are inferred representations of the extended function and the implementation respectively. In the same manner σ_{ext} is a substitution used in the inference of e_{ext}, and σ_{imp} is a substitution used in the inference of e_{impl}.

We check if *types* of r_{ext} and $r_{imp_L} \to r_{imp_R}$ unify. If they do, we add $r_{imp_L} \to r_{imp_R}$ to the set of representations.

A special case arises, if the extended function, does not have any implementation. In that case, the extended function infers to a representation in form $[r_1^*, r_2^*, \ldots, r_n^*] \to A$. This type-wise unifies or not unifies with $r_{imp_L} \to r_{imp_R}$, but we have no set to add the representation to. Therefore, we instead return the singleton $\{r_{imp_L} \to r_{imp_R}\}$.

Since the implementation e_{impl} is specific for certain representation, we cannot use its substitution σ_{impl} for merging. Such merge leads to a conflict, since the arguments of the implementations differ in representations. Therefore, we infer with the empty substitution.

We discuss the cost function now. We make sure that representation r_{cost} unifies with $r_{imp_L} \to$ `Int:Native`. If they do, the cost function has the correct type. If they do not, we return an error. You can see the complete pseudo code in Algorithm 5.

H. Application

The inference of the function application is more complicated than in other languages, due to the presence of extended functions and automatic representation conversions. First, we show an auxiliary algorithm, which is a variation upon the traditional application inference rule [1]:

$$\frac{f : A \to B, x : A}{f(x) : B}$$

Then, we proceed to the main algorithm, which takes Velka's specifics into account.

The auxiliary algorithm (see its pseudo-code in Algorithm 6) is used to infer the representation of the application result, along with the used substitution. It accepts

[2]We can safely assume, that the representation of e_{imp} have this form, since e_{imp} is a lambda expression by the definition.

Algorithm 6: Inferring result of a function application

1 **Function** APPLICATIONRESULTREPRESENTATION
2 ($[r_{a1}, \ldots, r_{an}] \to r_r$, σ_f, $[s_{a1}, \ldots, s_{am}]$, σ_a)
3 $\rho \leftarrow$ UNIFYREPRESENTATIONSASTYPES($[r_{a1}, \ldots, r_{an}]$, $[s_{a1}, \ldots, s_{am}]$;
4 **if** $\rho =$ false **then raise error**;
5 $\rho' \leftarrow \{A \backslash x | A \backslash x \in \rho$ and $A \notin \{r_{a1}, \ldots, r_{an}\}\}$;
6 $\phi \leftarrow \rho' \cup_S \sigma_f$;
7 **if** $\phi =$ false **then raise error**;
8 $\phi' \leftarrow \phi \cup_S \sigma_a$;
9 **if** $\phi' =$ false **then raise error**;
10 **return** $\langle \phi(\rho(r_r)), \phi \rangle$

$[r_{a1}, r_{a2}, \ldots, r_{an}] \to r_r$ the applicable representation of the function, $[s_{a1}, s_{a2}, \ldots, s_{am}]$ the representation of the arguments, σ_f the substitution of the function, and σ_a the substitution of the arguments.

We search for a type unifier ρ of $[r_{a1}, r_{a2}, \ldots, r_{an}]$ and $[s_{a1}, s_{a2}, \ldots, s_{an}]$ on line 3, to ensure type safety. We use Algorithm 4, since functions in Velka can be applied with arguments of the correct type and an arbitrary representation. We assume that an arbitrary conversion between representations exists. If ρ does not exists, we return an error.

We cannot use ρ in the further inference. It might introduce an incorrect inference on universally quantified type variables, that are part of the lexical closure. Consider the following example:

```
(define id (let-type (X) (lambda ((X x)) x)))
(tuple (id 42) (id #t))
```

We can easily see that the representation of the function `id` is $X \to X$. In the tuple expression we apply `id` with the `Int:Native` argument, getting $\rho = \{X \backslash \texttt{Int:Native}\}$. In the second application of `id` we get $\rho = \{X \backslash \texttt{Bool:Native}\}$ in the same manner.

We omit the rest of the algorithm for now. If we merge the ρ in the substitution of the whole tuple, the two inferred substitutions $X \backslash \texttt{Int:Native}$ and $X \backslash \texttt{Bool:Native}$ conflict. But that is not correct, since the type variable X in `id` is universally quantified. Thus such information is excluded from the substitution.

On line 5 we create a substitution ρ' as $\{A \backslash x \in \rho$ such that A is not an universally quantified variable in the $[r_{a1}, r_{a2}, \ldots, r_{an}] \to r_r\}$. This solves the aforementioned issue.

Substitution ϕ aggregates σ_a (the arguments inference), σ_f (the function inference) and ρ' ensuring the substitutions do not conflict.

The inferred representation is $\phi(\rho(r_r))$—an application of the original substitution ρ and the merged substitution ϕ on the right side of the function representation. Used substitution is ϕ.

The application $(f\ \ a_1\ \ a_2 \ldots a_n)$ consists of a function expression f and an argument tuple $[a_1, a_2, \ldots, a_n]$. The inference of the argument tuple yields the representation

Algorithm 7: Function application inference algorithm

```
1  Function INFERAPPLICATIONREPRESENTATION(e_f, [a_1, ..., a_n], E)
2    ⟨r_f, σ_f⟩ ← INFERREPRESENTATION(e_f, E);
3    ⟨[r_a1, ..., r_an], σ_a⟩ ← INFERREPRESENATION ([a_1, ..., a_n], E);
4    if r_f is a x_r → y_s then
5      return APPLICATIONRESULTREPRESENATION (x_r → y_s, σ_f, [r_a1, ..., r_an], σ_a)
6    else if r_f is a variable A then
7      Let B → C where B and C are new unused representation variables;
8      return APPLICATIONRESULTREPRESENATION (B → C, σ_f ∪ {A\B → C}, [r_a1, ..., r_an], σ_a)
9    else if r_f is in a form {x_t1 → y_u1, x_t2 → y_u2, ..., x_to → y_uo} then
10     Let {⟨x_ri, σ_i⟩ | ⟨x_ri, σ_i⟩ = APPLICATIONRESULTREPRESENTATION (x_ti → y_ui, σ_f, [r_a1, ..., r_an], σ_a)};
11     return ⟨{x_ri | i = 1, ..., o}, ∏_{i=1,...,o} (σ_i)⟩
12   else raise error;
```

$[r_{a_1}, r_{a_2}, \ldots, r_{a_n}]$ and the used substitution σ_a. The inference of the function yields the representation r_f and the used substitution σ_f. We discern three cases:

(i) The representation r_f has form $x_r \to y_s$. This is the simplest case. We directly use Algorithm 6 to get a representation and a substitution.

(ii) The representation r_f has form A, where A is a type variable. In this case f is either not bound, or it is a variable of unknown representation. We make new unused type variables B and C, and use $B \to C$ as the argument for algorithm 6. We also add a binding $A \backslash B \to C$ to the substitution σ_f, which is passed to Algorithm 6. This ensures the $A \backslash B \to C$ is passed to the resulting substitution, and the representation, for which A stands, is known.

(iii) In the last case r_f has a form $\{x_{t_1} \to y_{u_1}, x_{t_2} \to y_{u_2}, \ldots, x_{t_o} \to y_{u_o}\}$ of a representation set. We cannot discern which representation is used in the runtime, since the cost function is not evaluated in the inference phase. Therefore, we propagate all possible representations of the result in a representation set.

We call the Algorithm 6 with each representation $x_{t_i} \to y_{u_i}$. We collect the inferred representations to a set and aggregate the used substitutions by the substitution composition. You can see the resulting pseudo-code in the Algorithm 7.

VI. Experimental Evaluation

We conducted several preliminary experiments in order to measure the performance impact of suggested algorithms and concepts. All experiments were carried out by our Velka implementation [7]. This implementation compiles source code into Clojure [6] source code. This generated Clojure source was then used to run experiments on a computer with two Intel Xeons E5-2680, 64 GB RAM, Debian Linux, OpenJDK 11, and Clojure 1.10.

A. Sorting Implementation

We focused our experiments on an implementation of traditional sorting algorithms. For small data InsertSort algorithm should be faster than QuickSort. Hence, it may be reasonable to switch the sorting algorithm based on the size of input data. In our experiments, we sorted arrays of integers using two representations: `Array:Insertsort` and `Array:Quicksort`. Each representation had its own sorting algorithm using InsertSort and QuickSort, respectively, their detailed description can be found in [10]. The underlying data structure of both representations was a Java ArrayList.

There are three algorithms measured in our experiment. The first is a function *quicksort*, which is a simple function accepting an `Array:Quicksort` argument and uses Quicksort algorithm. The second is *insertsort*, a simple function accepting an `Array:InsertSort` and using InsertSort. The last algorithm—*sort extended* is an extended function accepting any `Array` implementation and using either QuickSort-like divide and conquer recursively calling itself, or InsertSort for small arrays. The divided sub-arrays for QuickSort are eventually sorted using InsertSort once they are small enough. In fact we implicitly obtained a hybrid algorithm.

The threshold for switching from QuickSort to InsertSort was obtained experimentally, by previous experiments with QuickSort and InsertSort algorithms. In the following experiments, the threshold was set to an array of 7 elements, i.e. arrays with 7 or less elements were sorted using InsertSort and larger arrays were sorted using QuickSort. Each algorithm is implemented using an iterative approach in order to get as much performance as possible.

B. Experiments and their results

We sorted arrays of randomly generated positive integers in our experiments. We pre-generated experimental data by a script that uniformly drew numbers ranging from 0 to 9999.

We conducted three batches of experiments to observe the algorithms running on different array sizes. The first batch of experiments is focused on small arrays, up to the 100 elements. The intention is to set the threshold for the *sort extended* algorithm and prove suitability of extended algorithms on a small scale. The second batch of experiments inspects medium-sized arrays ranging from 100 elements to 2900 elements. This experiment intends to compare all three algorithms on a scale, where each one runs in a reasonable time. The last batch of experiments inspects large arrays of integers. It sorted arrays ranging from $200,000$ elements to $500,000$ elements. It intends to compare the performance of large data. All experiments measure time to sort the array in milliseconds.

The results for small data are presented in Fig. 1. Small data were easily handled by each algorithm. Even on the small scale, InsertSort shows worse performance compared to QuickSort and sort extended. The comparison between QuickSort and sort extended is more interesting. You can see the detail of this comparison in Fig. 1 (bottom). We can see, that the two algorithms are very similar in performance.

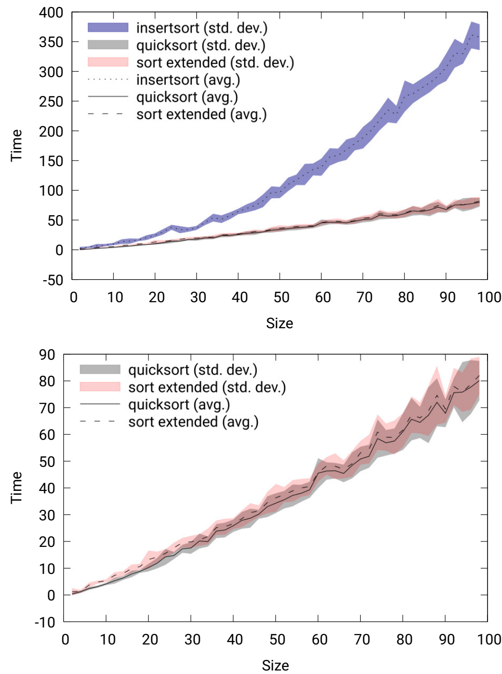

Fig. 1. Algorithm comparison for small data (top), detailed view (bottom)

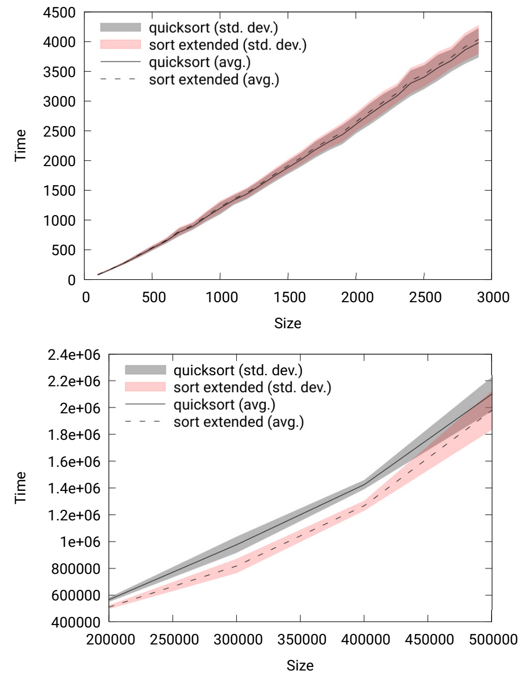

Fig. 2. Algorithm comparison for medium size (top) and large (bottom) data

QuickSort is winning by a small amount. This is probably due to the overhead of an extended function handling, used in sort extended. This suggests that even on a small scale an extended function can be used with only a limited performance loss.

You can see the results for medium-sized data in Fig. 2 (top). The performance of InsertSort is already hindered here. The two other algorithms are almost flat on the x-axis. Therefore, we omit results for InsertSort and present only details for QuickSort and sort extended. The performance gain of QuickSort over sort extended is diminishing. Although QuickSort still performs better, it seems that the overhead of extended functions is reduced due to the performance gain from switching algorithms. The InsertSort is not useful for larger data, and it seems that sort extended starts gaining performance wise on QuickSort and the trend continues.

Results for the experiments with large data are depicted in Fig. 2 (bottom). Comparison between QuickSort and sort extended shows significant development. Sort extended algorithm shows a performance gain around 10% on average compared to QuickSort. It seems, that switching algorithms outweighs overhead, caused by extended functions, on large scales.

VII. Conclusions and Future Research

The Velka language is a framework that allows to dynamically optimize data structures and algorithms to align with input data. We are to extend its standard library to provide various types and representations applicable in data analysis and data management. We are to reduce overhead of the Clojure language by targeting JVM directly. Additionally, we are to explore the possibility of supporting GPGPU computations for matrix and similar types. Further, we aim to research different strategies for setting and fine-tuning cost functions.

References

[1] B. C. Pierce, *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8

[2] C. J. Date, *An Introduction to Database Systems, Volume I, 5th Edition*. Addison-Wesley, 1990. ISBN 0-201-52878-9

[3] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database systems - the complete book*. Pearson Education, 2002. ISBN 978-0-13-098043-4

[4] H. Abelson, R. K. Dybvig *et al.*, "Revised report on the algorithmic language scheme," *High. Order Symb. Comput.*, vol. 11, no. 1, 1998. doi: 10.1023/A:1010051815785

[5] G. Steele, *Common LISP: the language, 2nd Edition*. Digital Pr., 1990. ISBN 0131556649

[6] R. Hickey, "The clojure programming language," in *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus*, J. Brichau, Ed. ACM, 2008. doi: 10.1145/1408681.1408682 p. 1.

[7] R. Skrabal. (2023) Velka source codes. [Online]. Available: https://github.com/Schkrabi/TypeSystem/blob/master

[8] J. W. Lloyd, *Foundations of Logic Programming, 2nd ed.* Springer, 1987. ISBN 3-540-18199-7

[9] R. Hindley, "The principal type-scheme of an object in combinatory logic," *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, 1969. [Online]. Available: http://www.jstor.org/stable/1995158

[10] D. E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. ISBN 0-201-03803-X