der Bundeswehr
Universität München

Forschungsinstitut CODE
Sichere Software-Entwicklung

23. Kolloquium

Programmiersprachen und Grundlagen der Programmierung

Vorläufiger Tagungsband

Feldkirchen-Westerham — 25.-28. September 2025

Stefan Brunthaler (Hrsg.)

Vorwort

Das 23. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2025) findet vom 25. bis zum 28. September 2025 in Feldkirchen-Westerham (Bayern, Deutschland) statt. Die Veranstaltung dient dem zwanglosen Austausch neuer Ideen und Ergebnissen zu den Themen des Entwurfs und der Implementierung von Programmiersprachen einerseits sowie den Grundlagen und den Methodiken des Programmierens andererseits.

Erstmalig 1980 von den Forschungsgruppen der Professoren Friedrich L. Bauer (TU München), Klaus Indermark (RWTH Aachen) und Hans Langmaack (CAU Kiel) abgehalten, findet das KPS alle zwei Jahre an unterschiedlichen Orten Deutschlands und Österreichs statt und bietet ein offenes Forum für alle interessierten deutschsprachigen Wissenschaftler:innen. Insbesondere sollen junge Doktorand:innen ermutigt werden, ihre Arbeiten einem fachkundigen Publikum vorzustellen und mit diesem zu diskutieren.

Vorherige Kolloquien:

2023	Vaals	RWTH Aachen
2021	Kiel	Uni Kiel
2019	Baiersbronn	DHBW Stuttgart
2017	Weimar	Uni Jena
2015	Pörtschach am Wörthersee	TU Wien
2013	Lutherstadt Wittenberg	Uni Halle-Wittenberg
2011	Schloss Raesfeld, Raesfeld	Uni Münster
2009	Maria Taferl	TU Wien
2007	Timmendorfer Strand	Uni Lübeck
2005	Fischbachau	LMU München
2004	Freiburg-Munzingen	Uni Freiburg
2001	Rurberg in der Eifel	RWTH Aachen
1999	Kirchhundem-Heinsberg	FernUni Hagen
1997	Avendorf auf Fehmarn	Uni Kiel
1995	Alt-Reichenau	Uni Passau
1993	Garmisch-Partenkirchen	UniBw München
1992	Rothenberge bei Steinfurt	Uni Münster
1989	Hirschegg	Uni Augsburg
1987	Midlum auf Föhr	Uni Kiel
1985	Passau	Uni Passau
1982	Altenahr	RWTH Aachen
1980	Tannenfelde im Naturpark Aukrug	Uni Kiel

September 2025 Stefan Brunthaler

Inhaltsverzeichnis

Angluerus: Data-based Hardware-Software Binding Through Rowhammer

DANIEL DORFMEISTER, Software Competence Center Hagenberg, Österreich

Modern industrial machinery comes with advanced control software. Although this control software typically contains valuable intellectual property, it often lacks effective protection. Consider, for example, obfuscation: copying to cloned machinery suffices to bypass protection. Prior work has investigated possibilities to bind software to a specific device such that the software cannot be used independently of the device. GlueZilla, for example, binds software to hardware by making the execution of the software depend on device-specific Rowhammer-induced bit flips. The protected software thus exhibits a different behavior when executed on a device it was not compiled for. We present Angluerus, which extends the idea of GlueZilla into the data domain. By aligning data with Rowhammer-induced bit flips, Angluerus is both more portable and able to tolerate unstable bit flips. To frustrate dynamic analysis, Angluerus uses debug blocking in combination with moving the Rowhammer part to a separate process. We evaluate Angluerus w.r.t. two properties: practicality and performance. Two case studies serve to demonstrate practicality. To evaluate performance, we use a matching subset of the SPEC CPU 2017 benchmarks and report an order of magnitude improvement over GlueZilla.

Author's Contact Information: Daniel Dorfmeister, Software Competence Center Hagenberg, Hagenberg, Österreich, daniel.dorfmeister@scch.at.

Code-copying compilation in production — An Experience Report

M. ANTON ERTL, TU Wien, Österreich

A code-copying compiler implements a programming language by concatenating code snippets produced by a different compiler for a different language. This technique has been used in Gforth since 2003, and we have solved various challenges: in particular, which code snippets are can be copied and what to do about the others; and challenges posed by changes in compilers. The performance of Gforth is similar to that of SwiftForth, a commercial system with a JIT compiler; the implementation effort is comparable to 1–2 targets for SwiftForth's JIT compiler.

Author's Contact Information: M. Anton Ertl, TU Wien, Wien, Österreich, anton@mips.complang.tuwien.ac.at.

Designing Real-time Mission-critical Systems with the TeamPlay Coordination Language

CLEMENS GRELCK, Friedrich-Schiller-Universität Jena, Deutschland

It is estimated that 98% of the world's computing devices operate in embedded or cyber-physical systems, where non-functional properties of program execution, such as energy (budgets), time (budgets), security or fault-tolerance can be as crucial as functional correctness. The rise of the internet-of-things with the edge-fog-cloud computing continuum and the increasing heterogeneity and parallelism of computing platforms rather solidify than change the need for resource-aware software. These developments create new challenges for software engineering.

We introduce the coordination language TeamPlay that introduces energy, time, security and fault-tolerance as first-class citizens into the software design process. Following the concept of exogeneous coordination, TeamPlay enforces a stringent software architecture with strict separation of concerns between operational detail and application-level design. We discuss selected aspects of the TeamPlay compiler and runtime environment as well as the scheduling and mapping problem that today's heterogeneous parallel architectures for the cyber part of cyber-physical systems create.

Author's Contact Information: Clemens Grelck, Friedrich-Schiller-Universität Jena, Jena, Deutschland, clemens.grelck@uni-jena.de.

Modular unification of unilingual pointer analyses to multilingual FFI-based programs

CHRISTIAN HAMMER, Universität Passau, Deutschland

Modular analysis of polyglot applications is challenging because flows of heap objects must be resolved across language boundaries. The state-of-the-art analyses for polyglot applications have two fundamental limitations. First, they assume explicit boundaries between the guest and the host language to determine inter-language dataflows. Second, they rely on specific analyses of the host and guest languages. The former assumption is impractical concerning recent advancements in polyglot programming techniques, while the latter disregards advances in pointer analysis of the underlying languages. In this work, we propose to extend existing pointer analyses with a novel summary specialization technique that unifies points-to sets across language boundaries. Our novel technique leverages combinations of host and guest analyses with minor modifications. We demonstrate the efficacy and generalizability of our approach by evaluating it with two polyglot language models: Java-C communication via Android's NDK and Java-Python communication in GraalVM.

Author's Contact Information: Christian Hammer, Universität Passau, Passau, Deutschland, Christian.Hammer@uni-passau.de.

Advanced Syntax Extensions with MOSTflexiPL

CHRISTIAN HEINLEIN, Hochschule Aalen (University of Applied Sciences), Germany
christian.heinlein@hs-aalen.de

MOSTflexiPL is a general-purpose programming language, whose syntax can be freely extended and customized by every programmer. Based on a small set of predefined operators, it is possible to define new operators with arbitrary syntax, which do not only cover prefix, infix, and postfix operators, but also control structures, type constructors, and declaration forms. While the author's KPS 2023 paper gave an overview of major concepts of MOSTflexiPL, this paper focuses on more advanced syntax extensions which are possible with MOSTflexiPL.

1 Introduction

MOSTflexiPL, which is an acronym for **mo**dular, **s**tatically **t**yped, **fl**exibly **ex**tens**i**ble **p**rogramming **l**anguage, is a general-purpose programming language, whose syntax can be freely extended and customized by every programmer. The logo used in the paper title shall express the extreme flexibility provided by the language allowing even fancy constructions unimaginable with conventional languages. (Therefore, it might be advisable to forget almost all familiar and seemingly necessary limitations of other languages to be able to fully recognize MOSTflexiPL's capabilities.)

A basic principle enabling that flexibility is: Everything is an expression, i. e., the application of an operator to subexpressions, where operators might possess any number of names and operands in an arbitrary order. Apart from well-known prefix, infix, and postfix operators, this also includes "circumfix" operators such as `(•)` (an operand depicted by the bullet sign • enclosed in parentheses), control structures such as `if•then•else•end`, declaration forms such as `•:•` (a name and a type separated by a colon), and so on. Another basic principle is, that the language provides only a small set of predefined operators covering arithmetic and logic operations as well as basic control structures, which can be used to define arbitrary new operators.

As the name indicates, the language is statically typed – which imposes numerous challenges with respect to the already mentioned flexibility –, and it is currently implemented by a compiler and a run-time system written in C++.

While the author's KPS 2023 paper [5] gave an overview of major concepts of MOSTflexiPL, the primary goal of this paper is to show examples of more advanced syntax extensions which are possible with MOSTflexiPL. To make the current paper self-contained, several parts of the previous paper, which are necessary to understand the advanced examples, are repeated.

2 Simple Operator Declarations

To give a first example, the following simple declarations define operators computing the square and the absolute value, respectively, of an integer value `x`, which can afterwards be applied using well-known mathematical syntax, e. g., 5^2 or $|2-7|^2$:

```
(x:int) "²" -> (int = x * x);
"|" (x:int) "|" -> (int = if x > 0 then x else -x end)
```

An operator declaration generally consists of a *signature*, an arrow and a *result declaration*, where the signature is a sequence of *names* and *parameter declarations*, while the result declaration consists of a *type* (the result type of the operator), an equality sign, and the *implementation* of the operator, enclosed in parentheses. A name is either a sequence of letters and digits starting with a letter (denoting exactly this sequence of characters, e. g., `abc`) or a sequence of arbitrary characters en-

closed in quotation marks (denoting this sequence of characters without the quotation marks, e. g., `"²"` denoting ²). A parameter declaration consists of a name, a colon, and a type, enclosed in parentheses. Finally, the implementation and the types mentioned above are – according to the basic principle mentioned in Sec. 1 – expressions. (At the moment, types are atomic expressions such as `int` or `bool`, but see Sec. 3 and Sec. 5 for more complex type expressions.)

When an operator application such as `|2-7|²` is evaluated at run time, the parameters of the operator are initialized from left to right by recursively evaluating the corresponding operands, and then the value of the expression is determined by evaluating the implementation of the operator.

In the examples above, the implementation of the square operator uses the predefined multiplication operator `•*•`, while the implementation of the abs operator uses the predefined change sign operator `-•` as well as the conditional operator `if•then•else•end` that returns, according to the truth value of its first operand, either the value of its second or its third operand.

The semicolon used to separate the two operator declarations is a simple predefined infix operator that evaluates its left and right operand and returns the value of the latter and, therefore, is typically used to denote sequential execution of subexpressions. But – again according to the basic principle mentioned in Sec. 1 – since declarations are expressions, too, the semicolon is also used to separate multiple declarations. In contrast to many other programming languages, however, semicolon must not be used at the end of a sequence of subexpressions, because it is an infix operator.

To give another example, the following declaration defines an operator that recursively computes the factorial of an integer value `n`, which can also be applied using well-known mathematical syntax, e. g., `5!` or `5²!`:

```
(n:int) "!" -> (int = if n <= 1 then 1 else (n-1)! * n end)
```

The particular challenge for the compiler with a declaration like this is to already recognize and accept the new syntax defined by the declaration inside of its own implementation to allow recursive applications of the operator.

3 Constants and Variables

A declaration of the form *`name : type = init`* declares a constant with the given name and type whose value is obtained by evaluating the initializer expression *`init`*, e. g., `N : int = 5²`. If the type is omitted, e. g., `N := 5²`, it is automatically deduced from the type of the initializer. If the initializer is omitted, the constant receives a unique new "synthetic" value that is different from every other value of the type. While this is of limited usefulness for numeric types such as `int`, it is crucial for variable types described below and for user-defined types described in the KPS 2023 paper [5].

For any type `T`, the type `T?` denotes memory cells containing values of type `T`. Therefore, a declaration such as `x : T?` defines `x` as a constant referring to a unique new memory cell that contains a value of type `T`, i. e., `x` actually denotes a variable with content type `T`. The current value contained in such a variable can be queried with the prefix question mark operator `?•`, and it can be changed with the assignment operator `•=!•`. The initial value of a variable is *nil*, which is a predefined value for any type denoting the absence of a real value, that is different from every other value of the type. Because a variable with content type `T` is itself a value of type `T?`, it might itself be stored in a variable of type `T??`. If the content of such a variable is queried prior to any assignment to the variable, the returned value is the nil value of type `T?`. If the content of this nil variable is queried in turn, it will be the nil value of type `T`, and assigning any value to such a nil variable has no effect.

(This behaviour is roughly comparable to reading and writing the Unix special file `/dev/null`: read operations return EOF, i. e., nothing, and write operations are discarded.)

By using variables and the predefined loop operator `while•do•end`, the factorial operator mentioned in Sec. 2 can also be implemented in a more procedural style:

```
(n:int) "!" -> (int =
    f : int?; f =! 1;
    i : int?; i =! 2;
    while ?i <= n do
        f =! ?f * ?i;
        i =! ?i + 1
    end;
    ?f
)
```

In the implementation of the operator, the variables `f` and `i` are declared and assigned their initial values as described above, where `i` is used as a loop counter running from `2` to `n`, while `f` accumulates the factorial value that is finally returned.

4 Optional, Alternative, and Repeatable Syntax Parts

To provide even more syntactic flexibility, the signature of an operator declaration might also contain optional, alternative, and repeatable parts using well-known EBNF syntax.

For example, the following declaration defines a variadic maximum operator that can be applied to any number of operands, e. g., `max of 1`, `max of 1 and 2`, `max of 1 and 2 and 3`, and so on:

```
max of (x:int) { and (y:int) } -> (int =
    m : int?; m =! x;
    { if y > ?m then m =! y end };
    ?m
)
```

According to EBNF, the curly brackets in the signature indicate that an application of this operator might contain the word `and` followed by an operand corresponding to the parameter `y` any number of times (zero or more). To access the different values of this parameter in the implementation of the operator, a corresponding curly bracket operator `{•}` is provided there, whose operand is repeatedly evaluated for every value of `y`. For the particular application `max of 1 and 2 and 3` this means, that the variable `m` declared in the implementation is initialized with the value of `x` (i. e., `1`), and then the `if` expression inside the curly brackets is evaluated in turn for `y` equal to `2` and to `3`, changing the value of the variable `m` to `2` and to `3`, respectively. Finally, the resulting value of `m` is returned.

To give another example, the following operator performs arbitrary calculations consisting of additions and subtractions, e. g., `calc minus 1 plus 2` or `calc 1 minus 2 plus 3`:

```
calc [minus] (x:int) { (plus|minus) (y:int) } -> (int =
    res : int?;
    res =! [-x | x];
    { res =! (?res + y | ?res - y) };
    ?res
)
```

In addition to the curly brackets denoting a repeatable part, the signature of this operator also contains square brackets denoting an optional part as well as round brackets containing two or more alternative parts separated by vertical bars. To find out in the implementation of the operator, whether the optional word `minus` after the word `calc` is present or not in a particular application of the operator, a corresponding square bracket operator `[•|•]` is provided, whose first or second operand, respectively, is evaluated accordingly. Similarly, a round bracket operator `(•|•)` with two operands corresponding to the round brackets with two alternatives in the signature is provided, whose first or second operand is evaluated according to whether the first or second alternative has been chosen in a particular application of the operator or – because in this example the round brackets are nested inside the curly brackets – in the respective pass through the curly brackets. The result value of a square or round bracket operator is the value of the operand that has been evaluated, while the result value of a curly bracket operator is the number of passes through these brackets. Taken together, these bracket operators allow the implementation of the operator to exactly determine the structure of a particular operator application and to process the values of its operands in a rather concise manner.

Generally speaking, all three kinds of brackets can have any number of alternatives separated by vertical bars, except that round brackets must contain at least two, because round brackets with just one alternative are useless. Therefore, the corresponding bracket operators provided in the implementation of the operator have a corresponding number of operands separated by vertical bars, where the *i*-th operand is evaluated if the *i*-th alternative has been chosen in a particular operator application or pass through curly brackets. As an exception, an operator corresponding to square brackets has an additional optional operand, that is evaluated (if it is present) if none of the alternatives has been chosen.

Therefore, the `calc` operator could also be defined as follows:

```
calc [minus] (x:int) { plus (y:int) | minus (z:int) } -> (int =
    res : int?;
    res =! [-x | x];
    { res =! ?res + y | res =! ?res - z };
    ?res
)
```

5 Generic Operators

If an optional parameter appears in the type of another parameter of the same operator, its value can be automatically deduced from the type of the operand corresponding to the other parameter and, therefore, the former parameter is called a *deducible parameter*. This can be used to define generic operators similar to C++ templates and Java generics, for example:

```
$$ Definition.
[(T:type)] (x:T?) "<->" (y:T?) -> (T? =
    z := ?x; x =! ?y; y =! z; y
);

$$ Application.
v1 : int?; v1 =! 1;
v2 : int?; v2 =! 2;
v1 <-> v2
```

Because `v1` and `v2` both have type `int?`, `v1 <-> v2` is a correct application of the previously defined swap operator, where the parameters `x` and `y` are initialized with the explicit operands `v1`

and v2, respectively, while the optional parameter T is implicitly initialized with the type int causing the type int? of the operands v1 and v2 to match the type T? of the corresponding parameters x and y. The implementation of this operator swaps the values contained in the variables x and y and returns the variable y.

6 Lambda Parameters

6.1 Problem

The following example shows an operator defining the syntax of for loops as well as a typical application of the operator:

```
$$ Definition.
for (var:int?) "=" (lower:int) ".." (upper:int) do
                                    [(B:type)] (body:B) end -> (int =
    var =! lower;
    while ?var <= upper do
        body;
        var =! ?var + 1
    end
);

$$ Application: Should print the numbers 1 to 10.
i : int?;
for i = 1 .. 10 do
    print ?i
end
```

According to Sec. 2, an application of this operator is evaluated by first initializing its parameters from left to right with the values of the corresponding operands, that means:

- The Parameter var is initialized with the variable i.
- The parameters lower and upper are initialized with the values 1 and 10, respectively.
- The deducible parameter B is initialized with the type bool of the subexpression print ?i.
- The parameter body is initialized with the value resulting from the evaluation of this subexpression, which prints the current value nil of the variable i (i. e., a blank line) and returns the bool value true.

Afterwards, the operator's implementation is evaluated, which means:

- Variable var (i. e., i) is assigned the value lower (i. e., 1).
- While the value of that variable is less or equal to upper (i. e., 10), the subexpression body; var =! ?var + 1 is evaluated repeatedly:
  - The evaluation of the parameter body simply yields its constant value true, i. e., it has no effect at all. In particular, it does *not* evaluate the subexpression print ?i.
  - The evaluation of the assignment var =! ?var + 1 increments the variable var (i. e., i) by 1, so that its final value after the last iteration will be 11.

In summary that means, that the above application of the for loop does not print the numbers 1 to 10, but just a blank line.

6.2 Solution

The problem with the above definition of the `for` operator is, of course, that the subexpression `print ?i` is evaluated *once before* the evaluation of the operator's implementation, instead of being evaluated *repeatedly during* the evaluation of this implementation.

This can be remedied by defining the parameter `body` as a *lambda parameter*:

```
for (var:int?) "=" (lower:int) ".." (upper:int) do
                              [(B:type)] (\ body -> (B)) end -> (int =
    var =! lower;
    while ?var <= upper do
        body;
        var =! ?var + 1
    end
);

i : int?;
for i = 1 .. 10 do
    print ?i
end
```

Omitting several technical details, this means:

- In the implementation of the `for` operator, there is a local operator `body` defined as `body -> (B)`.
- The implementation of this local operator is provided by the corresponding operand of an application of the operator, i. e., the subexpression `print ?i` in the above example.
- Because `body` is now an operator instead of a constant (as in Sec. 6.1), each evaluation of `body` inside the `while` loop causes a new evaluation of of this subexpression which will print the current value of the variable `i`.
- Because the variable `var` (i. e., `i`) is incremented in each iteration of the `while` loop, the above application of the `for` operator now in fact prints the numbers 1 to 10 instead of a single blank line.

In other words, operands corresponding to a lambda parameter are passed *unevaluated* and will be evaluated every time the lambda parameter is evaluated during the evaluation of the operator's implementation.

The backslash resembling a λ character is necessary to distinguish a lambda parameter from a regular parameter whose type is an operator type:

- An operand corresponding to a lambda parameter `\ body -> (B)` must be an expression with type `B`, which is used as the implementation of an implicitly generated local operator defined as `body -> (B)`.
- On the other hand, an operand corresponding to a regular parameter `body -> (B)` must be an expression that directly yields an operator of a compatible type, i. e., a parameterless operator with result type `B`. (This is described in more detail in the KPS 2023 paper [5].)

7 Lambda Parameters with Parameters

If a lambda parameter – which actually denotes an operator as described in Sec. 6.2 – has itself parameters, they are made visible in the operands corresponding to the lambda parameter, for example:

```
$$ Definition.
for i "=" (lower:int) ".." (upper:int) do
                    [(B:type)] (\ body (i:int) -> (B)) end -> (int =
    var : int?;
    var =! lower;
    while ?var <= upper do
        body ?var;
        var =! ?var + 1
    end
);

$$ Application: Prints the numbers 1 to 10.
for i = 1 .. 10 do
    print i
end
```

This example differs from the example given in Sec. 6.2 in several important details:

- The `for` operator defined here requires the fixed name `i` following the initial name `for` instead of an operand yielding an arbitrary variable of type `int?`. Therefore, no variable is required for applications of this operator.
- Instead, there is a local variable `var` defined in the operator's implementation.
- The lambda parameter `body` defined here has itself a parameter `i` with type `int` and, therefore, requires an operand of type `int` (actually `?var`) when being applied.
- The operand corresponding to the lambda parameter in the application of the `for` operator is `print i` instead of `print ?i`, where `i` is actually the parameter `i` of the lambda parameter which is visible in this operand.
- In each evaluation of the lambda parameter `body`, its parameter `i` receives the value of the corresponding operand `?var`, i. e., the current value of the variable `var`.
- Because each evaluation of `body` causes an evaluation of the corresponding operand `print i` with the respective value of the parameter `i`, the entire `for` loop will again print the numbers 1 to 10.

8 Passing Names to Operators

Of course, the fixed name `i` used to denote the current iteration value in the `for` operator defined in Sec. 7 is somewhat artificial.

To define a more realistic operator that can instead be used with an arbitrary user-defined name, it is necessary to pass that name to the operator. For that purpose, the fixed name `i` is replaced with a parameter named `#name` with the special type `sym`, that is in turn used as the name of the parameter of the lambda parameter `body`:

```
$$ Definition.
for ("#name":sym) "=" (lower:int) ".." (upper:int) do
                [(B:type)] (\ body (#name:int) -> (B)) end -> (int =
    var : int?;
    var =! lower;
    while ?var <= upper do
        body ?var;
        var =! ?var + 1
    end
);

$$ Applications.
for j = 1 .. 10 do
    print j
end;
for k = 1 .. 10 do
    print k
end
```

This requires some explanations:

- According to Sec. 2, names containing special characters such as `#` must be enclosed in quotation marks in a declaration, but used without the quotation marks in applications of the constant, parameter, or operator defined by the declaration.
- Therefore, `#name` in the declaration `#name:int` of the parameter of the lambda parameter `body` denotes an application of the parameter defined as `"#name":sym`.
- If this parameter would be defined as `name:sym`, the declaration `name:int` of the parameter of the lambda parameter `body` would be ambiguous because `name` could denote either an application of the parameter `name` with type `sym` or directly the name `name`. To avoid such ambiguities, the names of parameters that shall be used in the declaration of other parameters must contain at least one special character.
- Expressions with type `sym` – in particular applications of parameters with type `sym` – can be used instead of names in the signature of a constant, parameter, or operator declaration. Therefore, `#name:int` is in fact a correct parameter declaration.
- Operands corresponding to parameters with type `sym` can be names as described in Sec. 2 (e. g., `xyz`, `"N′"` etc.) or arbitrary expressions with type `sym`, in particular applications of other parameters with type `sym` whose operands are in turn either names or such expressions etc. Therefore, such operands eventually denote names.
- When an operator that has parameters with type `sym` is applied, applications of these parameters are replaced with the names denoted by the corresponding operands.

Therefore, the expression

```
for j = 1 .. 10 do
    print j
end
```

is processed at compile time as follows:

- The parameter named `#name` of the `for` operator is initialized with the name `j`.
- The parameters `lower` and `upper` are initialized with the values 1 and 10, respectively.
- Before the operand corresponding to the lambda parameter `body` is parsed, the application `#name` of the aforementioned parameter is replaced with the name `j`, causing the lambda parameter to actually have a parameter `j:int` which will be visible in the corresponding operand.
- Therefore, `print j` is a correct subexpression in this context (which would not be the case before or after the application of the `for` operator, because the parameter `j` is not visible there) which is used as the implementation of the lambda parameter.

At run time, the application of the `for` operator is evaluated in the same way as in Sec. 7.

9 Virtual Operators

9.1 Basic Principle

Applications of the following operator consist of an arbitrary type `T` followed by an arbitrary name, e. g., `int num` or `bool flag`, i. e., they denote declarations of variables `num` and `flag` with content type `int` and `bool`, respectively, in languages such as C and Java:

```
(T:type) ("#name":sym) -> (T? =
    #name : T?
)
```

In fact, the implementation of the operator actually contains a declaration of a variable with name `#name` and content type `T`, which are replaced with the name and type, respectively, which are passed to an application of the operator. This variable, however, is a local variable which is not visible outside of the operator's implementation.

To make it visible there, the above operator might be turned into a *virtual operator* by simply replacing the arrow `->` with a double-headed arrow `->>`:

```
(T:type) ("#name":sym) ->> (T? =
    #name : T?
)
```

Again omitting several technical details, an application of a virtual operator such as `int num` is replaced *at compile time* with the implementation of the operator, i. e., `#name : T?` in this example, where applications of the operator's parameters (`#name` and `T`) are in turn replaced with the corresponding operands (`num` and `int`, respectively), finally yielding the declaration `num : int?` which logically replaces the original expression `int num`. Therefore, the variable `num` defined by this declaration will be visible in the sequel:

```
int num;
num =! 1;
print ?num
```

9.2 Haskell-like Function Definitions

Haskell provides a very concise syntax for function definitions and applications, e. g.:

```
$$ Definition.
square x = x * x

$$ Application.
square 5
```

Doing the same with MOSTflexiPL's predefined declaration syntax is much more verbose:

```
$$ Definition.
square (x:int) -> (int = x * x);

$$ Application.
square 5
```

By omitting the parameter and result type of the `square` operator, which can be automatically deduced by the compiler, the definition becomes a little bit shorter:

```
square (x:) -> (= x * x)
```

To make it as concise as in Haskell, a virtual operator typically defined in a library can be used:

```
$$ Virtual operator defined in a syntax library.
("#func":sym) ("#par":sym) "="
                [(P:type) (R:type)] (\ impl (#par:P) -> (R)) ->> (=
    #func (par:P) -> (R = impl par)
);

$$ Application of the virtual operator
$$ to define function square.
square x = x * x;

$$ Application of the function square.
square 5
```

At compile time, the expression `square x = x * x` is recognized as follows as an application of the virtual operator defined before:

- Its parameters `#func` and `#par` are initialized with the names `square` and `x`, respectively.
- Before the operand `x * x` corresponding to the lambda parameter `impl` is parsed, `#par` is replaced with the name `x` in the declaration of its parameter, making the parameter `x` with type `P` (which is currently unknown, but will be deduced later) visible in this operand.
- Because `x` with unknown type `P` as well as the predefined multiplication operator `(int) "*" (int) -> (int)` are visible, `x * x` is recognized as a correct subexpression with type `int` by deducing `P` as `int` along the way.
- This subexpression is used as the implementation of the lambda parameter `impl` deducing `R` also as `int` along the way.

Because the operator is virtual, its application is replaced at compile time with its implementation, in which applications of the operator's parameters (`#func`, `P` and `R`) are in turned replaced by the corresponding operands, yielding the declaration

```
square (par:int) -> (int = impl par)
```

where `impl par` is an application of the following local operator that is implicitly generated for the lambda parameter `impl` using the operand `x * x` as its implementation:

```
impl (x:int) -> (int = x * x)
```

In summary, `square x = x * x` is replaced with a declaration of an operator `square (int) -> (int)` whose implementation computes the square of its parameter value and, therefore, this operator can be used in the sequel, e. g.:

```
square 5
```

9.3 Type Aliases

Another important use case of virtual operators, which cannot be achieved with normal operators, are type aliases.

If, for example, a programmer wants to define an alternative syntax `var of T` for variable types, because he dislikes the predefined syntax `T?`, this can be easily achieved with the following virtual operator:

```
$$ Definition.
var of (T:type) ->> (type = T?);

$$ Application.
x : var of int;
x =! 1;
print ?x
```

Because an application of a virtual operator is replaced at compile time with the operator's implementation where applications of the operator's parameters are in turn replaced with the corresponding operands, `var of int` is replaced with `int?` and, therefore, `x` has actually type `int?`.

If the operator `var of •` would be a normal, non-virtual operator, the expression `var of int` would still return the type `int?` at *run time*, but at compile it would be different from `int?` and, therefore, the expressions `x =! 1` and `?x` would not be type-correct, because they require an operand with type `T?` for some type `T`.

10 Outlook

During the last years, MOSTflexiPL has reached a certain degree of maturity and stability, which allows it to be employed for teaching purposes and for the implementation of small to medium-sized real world projects. In particular, the compiler messages in case of errors and ambiguities have been significantly improved during the last months. Furthermore, the compiler has been prototypically integrated into Microsoft's Visual Studio Code Editor and into the well-known text editor VI Improved (VIM) to make practical program development with MOSTflexiPL more convenient.

A topic that is neither covered by the KPS 2023 paper [5] nor by this paper are visibility declarations, which are required to define user-defined scoping rules and locally confined syntax extensions.

To make the language even more flexible and powerful, several features are still desirable:

- Meta-operators, which are required to pass the values of a repeatable parameter to another operator accepting repeatable parameters.
- User-defined literals, e. g., for types representing date and time values.

- User-defined whitespace and comments to allow any desired syntax for both block and line comments.
- Dynamic redefinitions of operators [4], which allow amongst other things strictly modular extensions of existing code and thus support unanticipated software evolution.
- Basic operators for parallel execution and synchronization, which can then be used to define more convenient and advanced constructs for parallel programming.

Some of these features have already been developed and prototypically implemented in earlier versions of the compiler, but have not been integrated into the current compiler yet.

11 Related Work

During the history of programming language development, the idea of an extensible programming language has appeared every now and then.

One of oldest and most well-known examples is Lisp [9] with its different dialects and flavors. Similar to MOSTflexiPL, Lisp does neither distinguish between operators and functions nor between predefined and user-defined operators/functions. By defining new functions – or macros, whose syntactic appearance is identical to that of functions – a programmer is actually extending the language all the time. Another parallel to MOSTflexiPL is the fact that language extensions are defined in the language itself, and that a very small language core is sufficient for that purpose. However, there are also essential differences: First of all, Lisp does not possess a static type system. Furthermore, Lisp expressions must always be parenthesized, which significantly restricts the possibilities for defining new syntax. Finally, MOSTflexiPL does not have a "procedural" macro engine, i. e., no user code will be executed at compile time in order to perform syntactic transformations. In contrast, the compile time transformations provided by virtual operators are completely declarative as well as statically type-safe. In summary, MOSTflexiPL has considerable advantages over Lisp (complete syntactic freedom and static type safety), while the deliberately omitted procedural macro facility has not been perceived as a major limitation yet.

Dylan [3] is a more modern language that has been strongly influenced by Lisp's ideas. It also supports syntactic extensibility in the language itself (actually in a rewrite macro system which is an integral part of the language). Even though the programmer has more freedom than with Lisp's simple s-expressions, there are also strict syntactic limitations which cannot be exceeded. In contrast, the operator concept of MOSTflexiPL offers virtually unlimited syntactic freedom. Apart from that, Dylan does not have a static type system either.

Many different languages, e. g., Haskell [7], Prolog [2], and Scala [8], allow the user to extend at least the syntax of expressions by defining new operator symbols. Since functional languages, just as MOSTflexiPL, do not distinguish between expressions and statements, the syntax of statements (e. g., control structures) becomes also extensible in principle. However, the syntax of types and declarations still remains fixed. In MOSTflexiPL, however, the basic principle "*everything* is an expression" is also applied to types and declarations, whose syntax can be extended by means of virtual operators.

An approach whose basic ideas and objectives are almost identical to that of MOSTflexiPL is "π – a Pattern Language" [6]. The concept called pattern there – which is "the only language construct in π" – directly corresponds to an operator in MOSTflexiPL: It possesses a syntax, composed of names (or symbols) and placeholders for operands, and an associated meaning corresponding to the implementation of a MOSTflexiPL operator. Thus, both approaches provide the same virtually unlimited syntactic flexibility that ultimately stems from the lack of any predefined grammar.

A significant difference and advantage of MOSTflexiPL over π is once again the static type system, since π is completely dynamically typed. In fact, the endeavour to reconcile extreme flexibility on the one hand with a maximum of static checkability on the other hand has been and still is one of the most ambitious challenges in the development of MOSTflexiPL.

Apart from that, MOSTflexiPL provides several other useful facilities not found in π, e. g., implicit and deducible parameters (where the latter are dispensable in a dynamically typed language) or visibility and exclude declarations which allow, amongst others, user-defined scoping rules and locally confined syntax extensions.

Finally, MOSTflexiPL might also be considered an adaptive grammar formalism [1, 10]. Because "everything is an expression," there is a single non-terminal symbol `X` denoting expressions. Every operator declaration induces a new production for `X` whose right hand side can be derived from the operator's signature by treating the operator's names as terminal symbols and replacing explicit parameters with the non-terminal `X`. The type information associated with the parameters and the result type of the operator can be added as grammar attributes. Visibility declarations control the set of currently active productions, while exclude declarations can be used to rule out some otherwise possible derivations.

12 Conclusion

MOSTflexiPL is a programming language currently under development whose syntax can be extended and customized by its users in a virtually unlimited way, where a rather small number of core constructs is sufficient to support a broad range of different programming styles. Therefore, it can be used, amongst others, as an extensible general purpose programming language, but also as a host language for developing domain-specific languages. It possesses a static type system and is implemented by a compiler and a run-time system written in C++.

References

[1] H. Christiansen: "A survey of adaptable grammars." *ACM SIGPLAN Notices* 25 (11) November 1990, 35–44.

[2] W. F. Clocksin, C. S. Mellish: *Programming in Prolog* (Fourth Edition). Springer-Verlag, Berlin, 1994.

[3] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

[4] C. Heinlein: "Global and Local Virtual Functions in C++." *Journal of Object Technology* 4 (10) December 2005, 71–93, http://www.jot.fm/issues/issue_2005_12/article4.

[5] C. Heinlein: "MOSTflexiPL – An Extremely Flexible Programming Language." In: T. Noll, I. Fesefeldt (eds.): *22. Kolloquium Programmiersprachen und Grundlagen der Programmierung*. Aachener Informatik-Berichte AIB-2023-02, Department of Computer Science, RWTH Aachen, 2023, 55–72.

[6] R. Knöll, M. Mezini: "π – a Pattern Language." In: *Proc. 24th Ann. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)* (Orlando, FL, October 2009). *ACM SIGPLAN Notices* 44 (10) October 2009, 503–521.

[7] S. Marlow (ed.): *Haskell 2010 Language Report*. HaskellWiki, 2010. http://haskell.org/definition/haskell2010.pdf

[8] M. Odersky: *The Scala Language Specification* (Version 2.9). Programming Methods Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), May 2011.

[9] G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.

[10] Wikipedia Contributors: *Adaptive Grammar*. https://en.wikipedia.org/wiki/Adaptive_grammar (2023-08-30)

Programming Joins

FRITZ HENGLEIN, University of Copenhagen, Dänemark

In programming one often has to implement queries such as joins on in-memory data. Joins, and relational queries in general, have been considered difficult to program efficiently. It turns out they are not.

We show that worst-case optimal joins, also for cyclic joins, are straightforward to program using a few basic programming techniques usually taught and learned in the first year of a computer science bachelor program: immutable dictionaries such as hash tries or radix trees, choosing the smallest set to iterate over when intersecting sets, and iterating over the variables in a join query in any nesting order.

We provide a novel proof of worst-case optimality by amortization, where the execution cost is charged to the output generated from input that is extended with ghost data. We furthermore show experimentally that widely used SQL engines generate asymptotically and practically inferior code on fundamental cyclic joins. For example, for triangle queries on certain sparse input relations, our 10-line standard Python program (or a corresponding 20 line Haskell program) executes 400 to 50.000 times faster than employing a variety of SQL engines.

This suggests that dispatching queries on program data to an SQL database engine is sometimes, and we suspect often, not a good idea for one of the motivations behind language-integrated querying: computational efficiency.

This is joint work with Changjun Li, Mikkel Kragh Mathiesen and Mads Rehof.

Author's Contact Information: Fritz Henglein, University of Copenhagen, Kopenhagen, Dänemark, fritz@henglein.com.

Towards Language Product Line Engineering Using Classic Compiler Generators

Teaching Old Horses New Tricks

THOMAS KÜHN, Martin-Luther University Halle-Wittenberg, Germany

In recent years, research on language product line (LPL) engineering has emerged. Building on the ideas of modular compiler construction and software product line engineering (SPLE), LPLs enable language users to choose and pick language features from which a corresponding compiler, interpreter and/or integrated development environment is composed. While several LPL engineering approaches showcased their general applicability to individual cases, most approaches rely on their own especially tailored language workbench. Hence, I aim to identify design principles and techniques in classic compiler generators that support the development of LPLs. In this talk, I present ongoing work creating an LPL for the Sample Programming Language (LAX) [22] using the compiler construction toolkit Eli [10] as a classic compiler generator.

Additional Key Words and Phrases: Compiler Construction, Software Product Lines, Language Product Lines

Most compilers for programming languages are developed using compiler construction tools (or language workbenches [7]) that provide various domain-specific specification languages, e.g., for grammars, attribute equations, or transformation rules, to generate almost all parts of the compiler. However, as everything is generated, reusing parts of a compiler is generally to either copying these definitions to another compiler specification or reusing an entire compilation phase or the whole back-end. To remedy this, researchers focused on modular compiler constructions, e.g., [9, 11, 17, 16], to establish more fine-grained reusable specifications. In the past decades, compiler construction tools and language workbenches included means for systematically reusing compiler specifications, e.g., [6, 19, 21, 4, 2, 8], ultimately leading to the notion of language components as "a reusable unit encapsulating a potentially incomplete language definition [comprising] the realization of syntax and semantics of a (software) language." [3, p. 243]. Building on the ideas of modular compiler construction and language components, more recently, researchers coined the term language product line* (LPL), e.g., [5, 24, 13], to denote the systematic development of a set of compilers for a family of languages by composing reusable language components. Adopting ideas from software product line (SPL) engineering, language users should be able to choose and pick needed language features[1] from which a corresponding compiler is constructed.

While several LPL approaches appeared over the years, their scope and the level of abstraction on which they operate differs. In detail, while many approaches compose languages on the concrete or abstract syntax level only, e.g. [24, 14, 12, 15], few include the composition of intermediate or machine code generation, e.g., [3]. Moreover, while some showed their approaches' applicability to general programming languages, e.g., [20, 13], many relied on their own specialized and tailored language workbenches, e.g. [20, 8, 2]. Thus, it is difficult to identify fundamental underlying principles and techniques for the modular development and composition of languages uncovered in recent years.

[1] Following Vacchi and Cazzola [18], language features are either language constructs, e.g., if then else, or language concepts (without concrete syntax), e.g., recursion.

Author's Contact Information: Thomas Kühn, Software Engineering and Programming Languages Group, Martin-Luther University Halle-Wittenberg, Halle, Saxony Anhalt, D-06120, Germany, thomas.kuehn@informatik.uni-halle.de.

To remedy this, in my ongoing work I aim to answer the following research questions:

- (RQ1) How suitable are classic compiler generators for the creation of an LPL of a programming language?
- (RQ2) What are fundamental principles and techniques for the modular development and composition of programming languages?
- (RQ3) What properties must a language component exhibit to be composable with another component?

In pursuit of answering these questions, I have started to develop an LPL of the Sample Programming Language (LAX) [22, Appendix A] using the classic compiler construction toolkit Eli [10]. Eli provides a rich set of domain-specific languages to specify all phases of a compiler, ranging from concrete and abstract syntax via attribute grammars and definition tables to code selection and code generation [10]. Moreover, its ability to integrate auxiliary C code makes it very flexible. Following the fine-grained iterative development approach proposed by Zimmermann, Kühn, and WeiSSbach [23], I started with an initial compiler covering the smallest LAX program and systematically implementing language features and extending the feature model in small increments.[2] Thus far, I have followed the iteration plan outlined in [kuehn2023kps] and employed one of the earliest additive SPL implementation technique, i.e., preprocessor annotations [1, Cha. 5.3], whereas previously implemented features as well as the initial compiler version shall not be changed during development.

In this talk, I provide insights into how the different specification languages permit or hinder modular compiler development. Moreover, I highlight the implementation techniques used to allow for extending the initial compiler with additional language components, focusing on the composability of the different domain-specific specification languages provided by the Eli toolkit. I will outline the development of the initial compiler version and the implementation of the first features, highlighting the employed extension points. In particular, I will not only focus on grammars for the concrete and abstract syntax but also on attribute grammars through to code generator specifications. Furthermore, for each compilation phase, I discuss when a corresponding specification or code fragment is composable.

This, in turn, will enable researchers and practitioners to use existing compiler generators for developing LPLs and extend them with composable specification languages for the individual phases. Overall, guiding practitioners towards more modular compilers or full-fledged language product lines.

Acknowledgments

I want to thank my supervisor Prof. Wolf Zimmermann for establishing the fine grained iterative development approach and fruitful discussions regarding compiler development. Moreover, I want to thank Edward Sabinus who reworked the iteratively developed LAX compiler, my work is based on. Last but not least, I thank Elisabeth Fritsch for the many discussions about the LAX LPL during her masters thesis.

References

[1] Sven Apel et al. Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, 2013. ISBN: 9783642375200.

[2] Arvid Butting et al. “Systematic composition of independent language features”. In: Journal of Systems and Software 152 (2019), pp. 50–69.

[2] This approach is a well-established SPL development approach denoted the reactive approach [1, Cha. 2.4].

[3] Arvid Butting and Andreas Wortmann. “Language Engineering for Heterogeneous Collaborative Embedded Systems”. In: Model-Based Engineering of Collaborative Embedded Systems: Extensions of the SPES Methodology. Cham: Springer International Publishing, 2021, pp. 239–253. ISBN: 978-3-030-62136-0. DOI: 10.1007/978-3-030-62136-0_11. URL: https://doi.org/10.1007/978-3-030-62136-0_11.

[4] Thomas Degueule et al. “Melange: a Meta-Language for Modular and Reusable Development of DSLs”. In: 8th International Conference on Software Language Engineering (SLE’15). Ed. by Davide Di Ruscio and Markus Völter. Pittsburgh, PA, USA: ACM, 2015, pp. 25–36.

[5] Benjamin Delaware, William Cook, and Don Batory. “Product lines of theorems”. In: 2011 ACM international conference on Object oriented programming systems languages and applications. 2011, pp. 595–608.

[6] Torbjörn Ekman and Görel Hedin. “The JastAdd System — Modular Extensible Compiler Construction”. In: Science of Computer Programming 69.1-3 (2007), pp. 14–26.

[7] Sebastian Erdweg et al. “Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future”. In: Computer Languages, Systems and Structures 44 (2015), pp. 24–47.

[8] Luca Favalli, Thomas Kühn, and Walter Cazzola. “Neverlang and FeatureIDE just married: Integrated language product line development environment”. In: 24th ACM Conference on Systems and Software Product Line (SPLC’20): Volume A-Volume A. 2020, pp. 1–11. DOI: 10.1145/3382025.3414961.

[9] Harald Ganzinger. “Increasing modularity and language-independency in automatically generated compilers”. In: Science of Computer Programming 3.3 (1983), pp. 223–278.

[10] Robert W. Gray et al. “Eli: A complete, flexible compiler construction system”. In: Communications of the ACM 35.2 (1992), pp. 121–130.

[11] Uwe Kastens and William M. Waite. “Modularity and reusability in attribute grammars”. In: Acta Informatica 31 (1994), pp. 601–627.

[12] Thomas Kühn and Walter Cazzola. “Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines”. In: 20th International Software Product Line Conference (SPLC’16). Beijing, China: ACM, 2016, pp. 50–59.

[13] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. “Choosy and Picky: Configuration of Language Product Lines”. In: 19th International Software Product Line Conference (SPLC’15). Ed. by Goetz Botterweck and Jules White. Nashville, TN, USA: ACM, 2015, pp. 71–80.

[14] Thomas Kühn et al. “A Metamodel Family for Role-Based Modeling and Programming Languages”. In: 7th International Conference Software Language Engineering (SLE’14). Lecture Notes in Computer Science 8706. Västerås, Sweden: Springer, 2014, pp. 141–160.

[15] Manuel Leduc, Thomas Degueule, and Benoit Combemale. “Modular Language Composition for the Masses”. In: Proceedings of 11th International Conference on Software Language Engineering (SLE’18), SLE 2018. Boston, MA, USA: ACM, 2018, pp. 47–59. ISBN: 9781450360296. DOI: 10.1145/3276604.3276622. URL: https://doi.org/10.1145/3276604.3276622.

[16] Marjan Mernik and Viljem umer. “Incremental Programming Language Development”. In: Computer Languages, Systems and Structures 31.1 (2005), pp. 1–16. DOI: 10.1016/j.cl.2004.02.001.

[17] Nathaniel Nystrom, Michael R Clarkson, and Andrew C Myers. “Polyglot: An extensible compiler framework for Java”. In: International Conference on Compiler Construction. Springer. 2003, pp. 138–152.

[18] Edoardo Vacchi and Walter Cazzola. “Neverlang: A Framework for Feature-Oriented Language Development”. In: Computer Languages, Systems & Structures 43.3 (2015), pp. 1–40. DOI: 10.1016/j.cl.2015.02.001.

[19] Markus Völter and Vaclav Pech. “Language Modularity with the MPS Language Workbench”. In: 34th International Conference on Software Engineering (ICSE’12). Zürich, Switzerland: IEEE, 2012, pp. 1449–1450.

[20] Markus Völter et al. “mbeddr: an Extensible C-Based Programming Language and IDE for Embedded Systems”. In: 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH’12). Tucson, AZ, USA: ACM, 2012, pp. 121–140. DOI: 10.1145/2384716.2384767.

[21] Guido H. Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. “Language Design with the Spoofax Language Workbench”. In: IEEE Software 31.5 (2014), pp. 35–43.

[22] William M Waite and Gerhard Goos. Compiler construction. Springer Science & Business Media, 1995. ISBN: 0-387-90821-8.

[23] Wolf Zimmermann, Thomas Kühn, and Mandy WeiSSbach. “(Almost) Agile Development of Verified Compilers”. In: 22. Kolloquium Programmiersprachen und Grundlagen der Programmierung. Ed. by Thomas Noll and Ira Fesefeldt. AIB-2023-03. 2023, pp. 176–185. URL: https://publications.rwth-aachen.de/record/972197/files/972197.pdf.

[24] Steffen Zschaler et al. “VML*–a family of languages for variability management in software product lines”. In: Software Language Engineering: Second International Conference, SLE 2009. Springer. Denver, CO, USA, 2010, pp. 82–102.

d2d = XML für Autoren

MARKUS LEPPER, semantics gGmbH, Deutschland

Die Codierung XML hat sich in vielen Bereichen als Standard für die Modellierung und Verarbeitung von Textdokumenten bewährt.

Auch semi-formale Fließtexte (Berichte, Bedienungsanleitungen, Gedichte, Romane, Briefe) können von automatischer Verarbeitung profitieren. D2d gibt den Autorinnen solcher Dokumente ein Format, welches mit wenig syntaktischem Ballast, auch ganz ohne technische Hilfsmittel und mit möglichst geringer Störung des kreativen Flows erlaubt, in gewohntem Schreibtempo formal-korrekte XML-Dokumente zu notieren.

Weiterhin sollen auch Domain-Experten für ihre Fachgebiete mit einfachen Mitteln verständliche Text-Typ-Definitionen erstellen können. Für beide Zwecke müssen sehr unterschiedliche avancierte Techniken kombiniert werden.

Author's Contact Information: Markus Lepper, semantics gGmbH, Berlin, Deutschland, post@mlepper.de.

Is this a Language Killed by Incremental Improvements? Python, Can We Help?

STEFAN MARR, University of Kent, United Kingdom

In the Python community, two major players having investing into the future of Python over the past years. Microsoft's Faster CPython team pushed ahead with impressive performance improvements for the CPython interpreter, which has gotten at least 2x faster since Python 3.9 and they have a baseline JIT compiler for CPython, too. At the same time, Meta is worked hard on making free-threaded Python a reality and bring classic shared-memory multithreading to Python, without being limited by the still standard Global Interpreter Lock, which prevents true parallelism.

Both projects delived major improvements to Python, and the wider ecosystem. So, it's all great, or is it?

In this talk, I'll discuss some of the aspects the Python core developers and wider community seem to not regard with the same urgency as it seems necessary from my personal perspective. Concurrency bugs scare me, and I strongly believe the Python ecosystem should be scared, too, or look forward to the 2030s being "Python's Decade of Concurrency Bugs". We'll start out reviewing some of the changes in observable language semantics between Python 3.9 and today, discuss their implications, and because of course I have some old ideas lying around, I'll propose a way forward. In practice though, this isn't a small well-defined research project. So, I hope to find people that might want to follow me down the rabbit hole of Python's free-threaded future.

Author's Contact Information: Stefan Marr, University of Kent, Canterbury, United Kingdom, s.marr@kent.ac.uk.

T3: Tree Transformation Tool

STEPHAN MITTE, Martin-Luther-Universität Halle-Wittenberg, Deutschland

Das Werkzeug T3 wird für die Transformation des abstrakten Syntaxbaums in eine Zwischensprache im Übersetzerbau-Werkzeug Eli entwickelt. Im Gegensatz zu anderen Werkzeugen generiert T3 eine Attributierte Grammatik, mit derer spezifizierte Transformationen auf dem abstrakten Syntaxbaum angewendet werden können. Zur einfachen Beschreibung dieser Transformationen wurde die Transformationssprache T2L - Tree Transformation Language entwickelt. Diese enthält allgemeine Sprachfeatures zur Berechnung der Baumtransformationen und kann zusätzlich auf die Attribute des abstrakten Syntaxbaums zugreifen, wodurch komplexe Transformationen definiert werden können.

Author's Contact Information: Stephan Mitte, Martin-Luther-Universität Halle-Wittenberg, Halle, Deutschland, stephan.mitte@student.uni-halle.de.

Ein Language-Server für Java-TX

RUBEN KRAFT, Baden-Wuerttemberg Cooperative State University, Germany
MARTIN PLÜMICKE, Baden-Wuerttemberg Cooperative State University, Germany

Um den Herausforderungen der zunehmenden Komplexität moderner Entwicklungsumgebungen gerecht zu werden, sind leistungsstarke Tools notwendig, die Entwicklerinnen und Entwickler bei der Nutzung von Sprachen mit Typinferenz unterstützen. Die Konzeption und der Aufbau eines Language Servers für Java-TX, einer auf Java basierenden, zwar statisch getypten aber ohne jede Typannotation auskommende Sprache, werden in dieser Arbeit behandelt. Der Server verwendet das Language Server Protocol (LSP), um gängige Integrated Development Environment (IDE)s wie Visual Studio Code und IntelliJ IDEA mit Funktionen wie Syntaxhervorhebung, Fehlerdiagnosen, Typinferenz, Quick-Fixes und Inlay-Hints zu versorgen. Die Architektur ist modular aufgebaut und nutzt LSP4J. Diese Integration ermöglicht es Entwickelnden, typlose Programme effizienter zu erstellen und zu debuggen. Am Ende werden die Herausforderungen, Designentscheidungen und mögliche Erweiterungen, wie Performanceverbesserungen und zusätzliche Editor-Features, besprochen.

1 Einleitung

1.1 Motivation

Java-TX ist eine statisch getypten Sprache, die es erlaubt, im Quellcode auf explizite Typangaben verzichtet. Der Compiler ermittelt die Typen während der Kompilierung automatisch. Das reduziert den syntaktischen Overhead.
Ein Problem dieser Herangehensweise ist, dass eingefügten Typen für den Entwickler während des Programmierens nicht direkt ersichtlich sind. Es ist daher notwendig, dem Entwickler die Typen, die der Compiler berechnet hat, zurückzugeben, um die Nachvollziehbarkeit zu verbessern. Wenn es mehrere gültige Typen gibt, ist es sinnvoll, eine Auswahlmöglichkeit zu schaffen, damit der Entwickler den passenden Typ bewusst auswählen kann.
Java-TX ist, im Gegensatz zu etablierten Sprachen wie Java, noch nicht in moderne Entwicklungsumgebungen integriert. In Java-TX jedoch müssen Entwickler oft manuell mit der Kommandozeile arbeiten und verzichten somit auf wichtige Entwicklungsunterstützung.
Ein Language Server soll diese Lücke schließen und Java-TX in moderne IDEs integrierbar machen. Dieser Server bietet Funktionen wie Syntaxhervorhebung, Typinferenzanzeige, Typauswahl und Fehlerdiagnosen. Die Absicht ist es, Java-TX in Bezug auf die Entwicklerfreundlichkeit auf das Niveau aktueller Programmiersprachen zu bringen, ohne die Vorzüge der typlosen Programmierung zu verlieren.

1.2 Zielsetzung

Die Funktionen, die in der Motivation vorgestellt wurden, sollen innerhalb eines zeitgemäßen, wiederverwendbaren Moduls umgesetzt werden. Hierbei wird besonders darauf geachtet, wie die Typinformationen, die der Compiler ermittelt, dargestellt werden, weil diese ein zentrales Merkmal und eine große Herausforderung von Java-TX sind.
Das Ziel ist es diese Erweiterung auf Basis des LSP zu bauen, um eine standardisierte und editorunabhängige Integration zu schaffen. Mit LSP ist es möglich, die entwickelte Lösung in verschiedenen Entwicklungsumgebungen zu nutzen.
Außerdem ist es wichtig, dass die Umsetzung eine geringe Latenz aufweist, wenn es um die

Authors' Contact Information: Ruben Kraft, Baden-Wuerttemberg Cooperative State University, Horb, Germany, rkraft@hb.dhbw-stuttgart.de; Martin Plümicke, Baden-Wuerttemberg Cooperative State University, Horb, Germany, pl@dhbw.de.

Bereitstellung von Typinformationen und anderen sprachspezifischen Funktionen geht. Es ist wichtig, dass Entwicklerinnen und Entwickler in Echtzeit Feedback bekommen, ohne dass ihre Arbeitsgeschwindigkeit dadurch beeinträchtigt wird.

2 Grundlagen

2.1 JavaTX - Eine typlose aber statisch getypte Sprache auf Basis von Java

Java-TX ist eine aus Java hervorgegangene Programmiersprache [4]. Sie ermöglicht es, Quellcode ohne typisierte Methoden oder Parameter zu schreiben. Während der Kompilierung ermittelt der zugrunde liegende Typinferenz- bzw. Typunifikationssalgorithmus die fehlenden Typen und setzt diese ein.
Java-TX vereint daher die Vorzüge von typlosen Sprachen, wie die hohe Ausdrucksstärke und Reduktion, mit der Sicherheit und Robustheit, die eine statische Typisierung bietet.
Außerdem weist Java-TX eine leicht angepasste Syntax im Vergleich zu klassischem Java auf, um typbedingte Konstrukte zu vermeiden oder zu vereinfachen. Alles in allem soll die Sprache den Entwicklungsprozess vereinfachen und beschleunigen, während sie dennoch die Vorteile der Typüberprüfung und einer sicheren Kompilierung nutzt.

2.2 Herausforderungen typloser Sprachen

Typlosen Programmiersprachen besitzen im Vergleich zu statisch typisierten Sprachen Vorteile. Hierzu gehören gesteigerte Flexibilität und verkürzte Entwicklungszyklen, da auf explizite Angaben von Typen im Quellcode verzichten wird.
Es ergibt sich eine Reduzierung der Entwicklungszeit, weil typbezogene Überlegungen wegfallen und die Programmlogik in den Vordergrund tritt.
Ein zentrales Problem typloser Sprachen zeigt sich jedoch in der mangelnden Transparenz während der Entwicklungszeit: Während der Entwicklungszeit, wissen Entwicklerinnen und Entwickler oft nicht, welche Typen der Compiler letztlich inferieren wird. Das macht es schwierig, das Programm zu analysieren und zu verstehen. Es wird besonders kritisch, wenn es mehrere gültige Typen gibt und der Entwickler bewusst auswählen muss.
Ein bedeutendes Ziel von typlosen, aber typsicherer Sprachen wie Java-TX sollte es daher sein, dass sie den Entwicklern geeignete Werkzeuge bieten, um inferierte Typen sichtbar zu machen. So kann man die Vorzüge von dynamischer und statischer Typisierung vereinen.

2.3 Das Language Server Protocol (LSP)

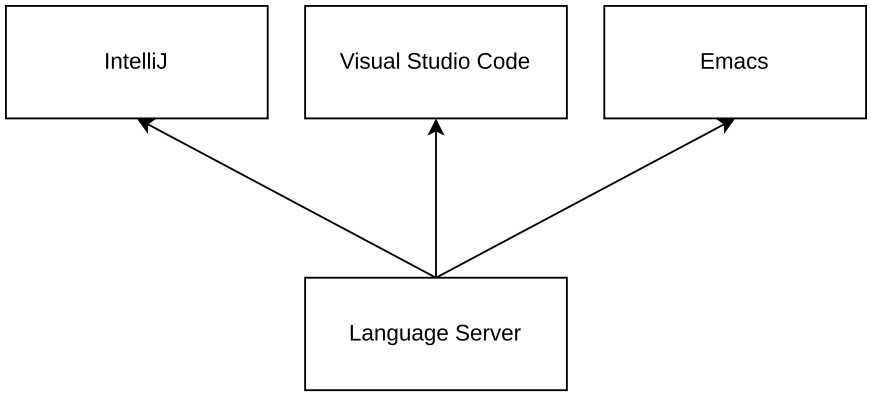

Fig. 1. Übersicht über Zweikomponenten-Architektur.

Plugins oder Extensions werden erstellt, um IDEs um neue Funktionen zu erweitern. In der Vergangenheit war es notwendig, für jede IDE eine eigene Erweiterung zu erstellen, weil sich die internen Architekturen und die Erweiterungsschnittstellen teils erheblich unterscheiden. Die Microsoft Corporation hat das LSP eingeführt, um die Wiederverwendbarkeit zu verbessern [3]. Es schafft eine weitere Abstraktionsebene zwischen der IDE (Client) und der sprachspezifischen Logik (Server). So kann ein Language Server für mehrere Clients und somit IDEs genutzt werden, ohne dass die Logik des Servers mehrfach entwickelt werden muss. Nur der Client muss an die Umgebung angepasst werden. Das Protokoll sieht vor, dass die Logik von der Anzeige getrennt wird. Dabei entstehen zwei Komponenten. Diese sind in Abbildung 1 dargestellt.

(1) Der Language-Server: Der Langauge Server enthält alle Logik und kommuniziert mit dem Client durch das LSP definierte Nachrichten.
(2) Der Client: Der Client enthält lediglich die konkrete Anzeigelogik für die IDE. Dabei besitzt jede IDE einen eigenen Client.

Generische LSP-Client-Implementierungen oder Frameworks ermöglichen eine schnelle Erstellung eines Clients. Die Kommunikation läuft dabei über LSP. Das erlaubt eine einheitliche und wiederverwendbare Interaktion, unabhängig von der spezifischen Entwicklungsumgebung.

2.3.1 Funktionsweise von LSP. Um ein Verständnis über das LSP zu bekommen, erfordert es zunächst, die Funktionen zu betrachten, die das Protokoll bietet. Das LSP legt eine standardisierte Schnittstelle fest, die es einem Editor (Client) und einem sprachspezifischen Server (Language Server) ermöglicht, miteinander zu kommunizieren. Das Ziel ist es, den Entwicklern Informationen ihres Programmcodes zu liefern, beispielsweise Fehler oder inferierte Typen.

Anschließend werden die wichtigsten LSP Funktionen erläutert.

(1) Inlayhints: Inlay Hints sind Hinweise, die im Quellcode angezeigt werden. Diese zeigen Informationen, wie etwa inferierte Typen an. Im Falle von Java-TX sind sie wichtig, um Entwicklern die durch den Compiler ermittelten Typen darzustellen, ohne dass der Quellcode verändert wird. Deswegen sind sie oft unter dem Namen Type Hints bekannt.

```
public class Test{
    public java.lang.Integer method1(){

        return 1;
    }
}
```

Fig. 2. Beispielhaftes Inlayhint

(2) Diagnostics: Eines der wichtigsten Merkmale des LSP sind die Diagnostics. Die meist durch Unterstreichung dargestellten Diagnostiken machen auf Fehler, Informationen oder zusätzlichen Hinweisen aufmerksam. Dabei werden verschieden Farben verwendet.

Fig. 3. Beispielhaftes Diagnostic

(3) Text-Edits: Text-Edits werden verwendet, um auf mögliche Anpassungen am Quellcode hinzuweisen. Sie beinhalten unter anderem Quick Fixes, die fehlende Imports ergänzen, Typen hinzufügen oder andere Verbesserungen vorschlagen. Werden die TextEdits eingefügt, so verändert sich der Programmcode.

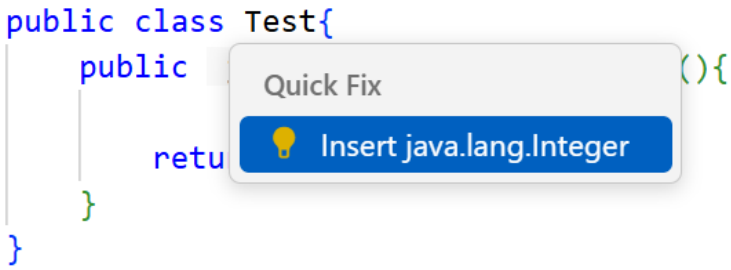

Fig. 4. Beispielhafter Quick-Fix

(4) Hovers: Wenn man mittels des Mauszeiger über bestimmte Codepositionen fährt, erscheinen zusätzliche Informationen. Diese sind beispielsweise die Typdefinition einer Variablen oder eine Dokumentation. Diese Informationen verbessern das Verständnis des Codes, ohne dass er explizit nachgeschlagen werden muss.

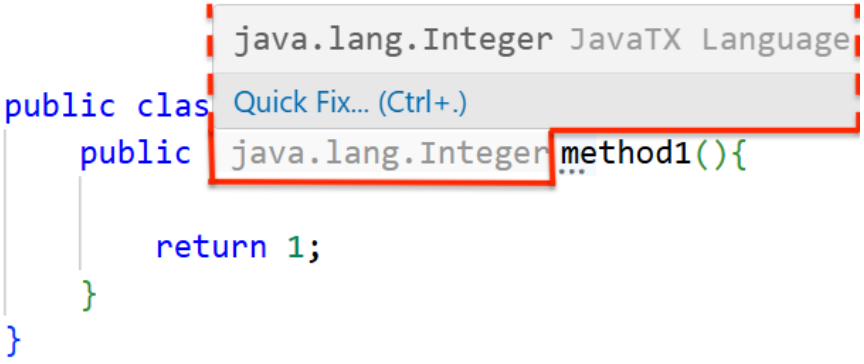

Fig. 5. Beispielhafter Hover-Effekt

Über das standardisierte JSON-RPC-Protokoll (Version 2.0) erfolgt die Kommunikation zwischen Client und Server, wobei alle Nachrichten als strukturierte JSON-Objekte gesendet werden. Nachrichten enthalten typischerweise diese Felder:

(1) `jsonrpc`: Zeigt die JSON-RPC-Version an, die verwendet wird. Bei LSP ist "2.0" der Standard.
(2) `id`: Eine eindeutige Kennung, um Anfragen und Antworten miteinander zu verbinden.
(3) `method`: Der Name der Methode, die ausgeführt werden soll, wie z. B. textDocument/hover.
(4) `params`: Ein Objekt, welches die Parameter für die Methode beinhaltet. Je nach Kontext variieren sie und sind oft der umfangreichste Teil der Nachricht.

```
Content-Length: ...\r\n
\r\n
{
        "jsonrpc": "2.0",
        "id": 1,
        "method": "textDocument/completion",
        "params": {
                ... //Parameter je nach Methode
        }
}
```

Listing 1. Beispiel für JRPC

Jede Nachricht im Protokoll erhält zudem einen Header mit dem Feld Content-Length, der die Größe der Nachricht festlegt. Ein komplettes Beispiel befindet sich in Listing 1. Nachdem der Client die Anfrage verschickt hat, bearbeitet der Language Server die übergebene Methode und sendet eine passende JSON-RPC-Antwort zurück. Je nachdem, welche Methode verwendet wird, wird im Language Server unterschiedliche Funktionalität ausgeführt. Eine Vielzahl von standardisierten Methoden werden in der LSP-Spezifikation definiert, um die Kommunikation zwischen Client und Language Server zu ermöglichen. Einige der wichtigsten und für diese Arbeit relevanten Methoden sind im Folgenden beispielhaft aufgeführt:

(1) `sendMessages`: Diese Methode ermöglicht es dem Language Server, Nachrichten direkt an den Client zu übermitteln. Dabei können sowohl allgemeine Statusinformationen als auch konkrete Hinweise, wie beispielsweise Fehlermeldungen oder Benachrichtigungen über erfolgreiche Kompilierungen, übertragen werden.
(2) `inlayHint`: Mithilfe dieser Methode werden Inlay Hints an den Client gesendet. Diese enthalten Informationen wie berechnete Typen oder ergänzende Hinweise und werden direkt im Quelltext an den entsprechenden Positionen angezeigt. Für die in dieser Arbeit entwickelte Erweiterung stellt diese Methode eine zentrale Grundlage dar, da die Typinferenz-Ergebnisse auf diesem Weg an den Entwickler übermittelt werden.
(3) `hover`: Diese Methode ermöglicht es dem Client, detaillierte Zusatzinformationen über eine bestimmte Codeposition abzurufen. Wird der Mauszeiger beispielsweise über eine Variable oder Methode bewegt, sendet der Client eine Anfrage an den Server, der daraufhin weiterführende Informationen wie Typdefinitionen, Dokumentationen oder Aufrufhierarchien zurückliefert.

Diese und viele weitere Methoden sind Teil der offiziellen LSP-Spezifikation und bilden die Grundlage für die standardisierte Kommunikation zwischen dem Language Server und den unterstützten IDEs. Insbesondere die Methoden `inlayHint` und `hover` sind essenziell für die Integration der in dieser Arbeit umgesetzten Typinferenzmechanismen.

2.3.2 LSP4J - LSP in Java. Das LSP stellt eine Spezifikation dar, aber es ist keine konkrete Implementierung. In der Programmiersprache Java gibt es mit LSP4J eine offizielle Implementierung, die von der Eclipse Foundation zur Verfügung gestellt wird [1]. Die grundlegende Kommunikation zwischen Client und Server wird von LSP4J übernommen. Es trennt die Protokolllogik von der Anwendungslogik. Indem es vordefinierte Interfaces bereitstellt, abstrahiert es die standardisierten LSP-Nachrichtenformate. Es gibt eigene Methoden innerhalb der Interfaces, die je nach dem welche Nachricht durch den Client an den Server gesendet wurde ausgeführt werden. Die Rückgabewerte der Methoden werden wieder an den Client übermittelt. Die Parameter werden in Java Objekte umgewandelt. Eine solche Methode kann in Listing 2 nachvollzogen werden. Beispielhaft wird hier die Methode für die Anforderung der InlayHints verwendet. Der Rückgabetyp beinhaltet eine Liste an Inlayhints. Das vereinfacht die Entwicklung erheblich und erlaubt es gleichzeitig, auf stabile, getestete Kommunikationspfade zurückzugreifen. Ein weiterer Vorteil von LSP4J in diesem Zusammenhang ist, dass der Compiler von Java-TX ebenfalls in Java verfasst ist. So ist es möglich, dass die Language Server und der Compiler direkt miteinander kommunizieren, ohne dass es weitere Übersetzungs- oder Integrationsschichten gibt.

```
1 @Override
2 public CompletableFuture<List<InlayHint>> inlayHint(InlayHintParams
      params) {
3     ...
4 }
```

Listing 2. Beispielmethode von LSP4J

Der Datenaustausch erfolgt standardmäßig über die Standard-Ein- und -Ausgabe. LSP4J unterstützt jedoch auch andere Transportprotokolle wie TCP oder UDP. Um einen voll funktionsfähigen Language Server zu erstellen, müssen innerhalb von LSP4J verschiedene Komponenten implementiert werden, die jeweils unterschiedliche Funktionen bieten, etwa für Textverarbeitung, Diagnostik oder Code-Aktionen.

3 Projektüberblick

Der Language Server, der hier vorgestellt wird, hat als Hauptzweck, dem Entwickler Typinformationen anzuzeigen und ihm die Wahl zwischen mehreren möglichen Typen zu lassen. Außerdem ist es das Ziel des Language Servers, dass er syntaktische Fehler in Java-TX erkennt und sie visuell hervorhebt. Weil die reguläre Java-Syntax in fast allen modernen Entwicklungsumgebungen standardmäßig unterstützt wird, führen die spezifischen Sprachkonstrukte von Java-TX häufig dazu, dass sie innerhalb der IDE fehlerhaft markiert werden.

3.1 Überblick über die Hauptkomponenten

Die gesamte Extension setzt sich aus drei klar abgegrenzten Komponenten zusammen, die zwar miteinander kommunizieren, aber funktional unabhängig sind:

(1) **Compiler-Interface:** Das Compiler-Interface ist ein Bestandteil des Java-TX-Compilers und stellt die Verbindung zu ihm her. Es erlaubt den Zugriff auf die vom Compiler erstellten ResultSets und den abstrakten Syntaxbaum (AST), den der Compiler erzeugt hat. Außerdem ist das Interface dafür zuständig, aus dem vollständigen, typisierten Quellcode den Bytecode zu erstellen.

(2) **Language Server:** Die zentrale Business-Logik der Anwendung wird im Language Server verwaltet. Er untersucht die Informationen, die der Compiler bereitstellt, legt die Positionen für diagnostische Hinweise fest und liefert Typhinweise (Inlay Hints). Dank dieser Abstraktion können ihn verschiedene Entwicklungsumgebungen entkoppelt genutzt werden.
(3) **Clients:** Die Clients agieren als Darstellungsschicht und kommunizieren über das LSP mit dem Language Server. Sie empfangen und fragen die Informationen, die der Server liefert, wie etwa Typhinweise, Fehlermeldungen oder Quick-Fixes ab und visualisieren diese innerhalb der jeweiligen IDE.

3.2 Unterstützte Entwicklungsumgebungen

Selbst wenn die Clients meist auf bestehenden Frameworks beruhen, erfordert ihre Entwicklung dennoch einen gewissen Aufwand. Die jeweiligen Client-Erweiterungen müssen den Language Server initialisieren und seinen Lebenszyklus verwalten. Obwohl diese Aufwände normalerweise gering sind, benötigen sie dennoch Tests und Wartung bei zukünftigen Änderungen.
In Anbetracht dieser Einschränkungen wurde die Entscheidung getroffen, nur eine Auswahl der gängigen Entwicklungsumgebungen zu unterstützen. Es wurden gängige und praxisnahe IDEs ausgewählt, um eine hohe Nutzbarkeit zu gewährleisten.
Die Entwicklungsumgebungen, die ausgewählt wurden, sind:

(1) **Visual Studio Code:** Visual Studio Code ist ein populärer Editor, der vor allem für seine hohe Erweiterbarkeit bekannt ist. Diese Umgebung wurde als Hauptzielplattform ausgewählt, weil sie eine große Nutzerbasis hat und Extensions leicht integriert werden können.
(2) **IntelliJ IDEA:** Die IDE IntelliJ IDEA gehört zu den meistgenutzten Optionen für Java [2]. Durch die einfache Installation von Plugins und die große Verbreitung im Java-Umfeld ist sie eine wichtige Zielplattform für Java-TX.
(3) **Emacs:** Emacs ist im Linux-Umfeld ein weitervererbter konfigurierbarer Editor, der insbesondere im Programmiersprachenforschung häufig eingesetzt wird. Er kann über `lsp-mode` mit der Language Server verbunden werden.

4 Funktionalität des Language Servers

4.1 Syntaxhighlighting

```
import java.lang.String;
import java.lan
import java.lan
```

extraneous input 'Test' expecting {'extends', 'implements', 'permits', '{', '<'}

View Problem (Alt+F8) No quick fixes available

```
public class t Test{
    public  java.lang.Comparable<? extends java.lang.c… method1(Boolean decider){

        if(decider){
            return "Hello World!";
        }
        return 1;
    }
}
```

Fig. 6. Syntaxcheck innerhalb der IDE

Das Syntaxhighlighting hebt fehlerhafte Syntax hervor. Fehler, welche durch das Parsen von ANTLR erhalten werden, werden unterstrichen und angezeigt (vgl. Abbildung 6). Die durch ANTLR enthaltenen Fehlerbeschreibungen werden angezeigt. Da Java-TX zwar eine Java-ähnliche Syntax besitzt, sie jedoch ohne Typen arbeitet, ist es essentiell dieses Feature zu unterstützen. Dadurch kann eine sinnvolle Arbeit mit Java-TX in modernen IDEs garantiert werden. Wenn dieses Feature nicht existieren würde, würden gar keine Syntaxfehler erkannt, oder eine Syntaxerkennung der Java-Syntax stattfinden. Ein Syntaxfehler ist in Abbildung 6 dargestellt.

4.2 Typehints und Typanzeige

Die vom Typinferenzalgorithmus ermittelten Typen werden dem Entwickler als Typhinweise direkt vor den entsprechenden Variablen angezeigt. Es ahmt die Typdeklarationen der klassischen Java-Syntax nach, was die Lesbarkeit des Codes verbessert, ohne dass der Entwickler die Typen ausdrücklich angeben muss. Die berechneten Typen werden zusätzlich als Informations-Diagnostiken unterhalb der Variablen dargestellt, um eine konsistente und redundante Darstellung der Ergebnisse zu gewährleisten. Es werden nur valide Inlayhints angezeigt. Falls eine Variable mehrere mögliche Typen besitzt, werden sie im Inlayhint zusammen aufgeführt. Die verschiedenen Möglichkeiten werden durch das Zeichen | getrennt. Außerdem wird für jeden einzelnen Typ eine eigene Diagnostik erstellt, damit der Entwickler eine gezielte Auswahl treffen kann. Außerdem ermöglicht die Erweiterung das automatische Einfügen eines vorgeschlagenen Typs über einen Quick-Fix in den Programmcode.

4.3 Einfügen von Typen

Fig. 7. Inlayhints mit Diagnostiken.

Ein Quick-Fix wird automatisch für jeden möglichen Typ einer Variablen erstellt, den der Language Server ermittelt. Mit dieser Funktion kann der Entwickler einen konkreten Typ direkt im Programmcode einfügen, ohne dass er ihn manuell eingeben muss.

Wenn man einen Quick-Fix auswählt, wird nicht nur der passende Typ in den Code eingefügt, sondern auch die interne Logik zur Typauswahl aktualisiert. Das heißt: Wenn man einen konkreten Typ für eine Variable festlegt, werden alle nun ungültigen Typkombinationen ausgeschlossen. So wird garantiert, dass nur gültige und konsistente Typkombinationen für die verbleibenden Typplatzhalter verbleiben. Die inhärente Typabhängigkeit innerhalb der vom Compiler erstellten ResultSets ist der Grund für dieses Verhalten: Wenn man einen Typ auswählt, beeinflusst das die Gültigkeit der anderen Typzuweisungen. Eine Auswahl und die Inlayhints können in Abbildung 7 nachvollzogen werden.

Der Quellcode wird beim Einfügen direkt verändert, im Gegensatz zu den rein informativen Inlay-Hints, die nur zusätzliche Informationen bieten, aber keinen Einfluss auf den Programmtext haben.

4.4 Fehlererkennung und Diagnostik

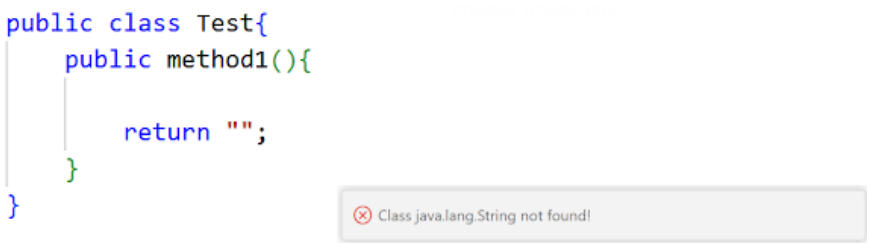

Fig. 8. Fehlermeldung bei fehlenden Imports.

Wenn ein Programm nicht erfolgreich mit dem Java-TX-Compiler kompiliert werden kann, wird eine Fehlermeldung erstellt, als Nachricht an den Client gesendet wird. Die genaue Beschreibung des Kompilierungsfehlers werden in dieser Meldung angezeigt.
Typische Ursachen für Fehler sind zum Beispiel fehlende Import-Anweisungen für bestimmte Typen oder nicht auflösbare Referenzen. Dies kann in Abbildung 8 gesehen werden. Indem diese Fehler direkt angezeigt werden, hat der Entwickler die Möglichkeit, genau auf die Ursache zu reagieren und entsprechende Anpassungen vorzunehmen. Das ist ein wichtiger Faktor für die Effizienz der Entwicklungsarbeit mit Java-TX.

4.5 Live-Compilation und Ausgabe

Sobald der Programmcode in der Entwicklungsumgebung gespeichert wird, initiiert das Compiler-Interface automatisch einen Kompilierungsvorgang. Durch diesen Prozess wird ausführbarer Bytecode erstellt, der als `.class`-Dateien ausgegeben wird.
Die Dateien, die erstellt werden, landen im gleichen Verzeichnis wie die Quelldateien, strukturiert in einem separaten `/out`-Ordner. Dies kann in Abbildung 9 gesehen werden. In diesem Ordner sind alle kompilierten Klassen abgelegt, sodass das Programm ohne weiteren manuellen Eingriff direkt ausgeführt werden kann.

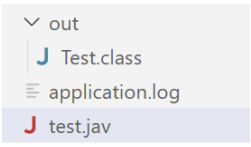

Fig. 9. Bytecode innerhalb des /out Ordners.

Durch die Automatisierung der Kompilierung ist kein manuelles Kompilieren über die Konsole notwendig. Das vereinfacht den Entwicklungsprozess mit Java-TX erheblich und sorgt dafür, dass die Sprache gut in moderne Entwicklungsumgebungen integriert ist.

5 Architektur des Systems

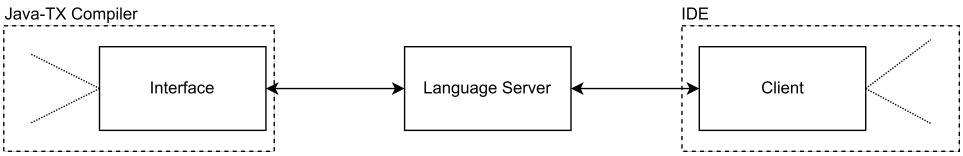

Fig. 10. Gesamtarchitektur des Systems.

Die Architektur des Language Servers ist insgesamt aufgebaut auf den drei Hauptkomponenten, die vorher beschrieben wurden. Diese können in Abbildung 10 nachvollzogen werden. Das Compiler-Interface, das Teil des Compilers ist, wird als Abhängigkeit dem Language Server hinzugefügt und kommuniziert innerhalb der Java-Anwendung direkt mit ihm.
Der Language Server ist das Bindeglied zwischen dem Compiler-Interface und den unterschiedlichen Clients. Er kümmert sich um die Aufbereitung und Weiterverarbeitung der Daten, die der Compiler liefert. Um die Zugriffe auf das Compiler-Interface zu minimieren, da der Java-TX-Compiler und somit auch das Interface eine geringe Laufzeitperformance haben, werden empfangene Daten im Server zwischengespeichert und nur beim Speichern aktualisiert. Der Language-Server versucht die vorhandenen Daten korrekt anzupassen.
Die Clients, tauschen über `System.in` und `System.out` Nachrichten mit dem Language Server aus. Die standardisierten LSP-Nachrichten, die Informationen wie Typhinweise, Diagnostiken oder Textänderungsvorschläge umfassen, werden über diesen Kanal übertragen.

5.1 Komponenten im Detail

Anschließend werden die einzelnen Komponenten im Detail beschrieben und Designentscheidungen erläutert.

5.1.1 Compiler-Interface. Im Java-TX-Compiler ist das Compiler-Interface integriert. Es erlaubt das Kompilieren einer Quellcodedatei basierend auf ihrer URI, normalerweise der Datei, die derzeit in der IDE geöffnet ist. Falls erforderlich, werden weitere abhängige Dateien zusätzlich mitkompiliert.
Ein Transferobjekt wird nach dem Kompilierungsvorgang an den Language Server übermittelt. Es umfasst den abstrakten Syntaxbaum, in dem spezielle Typplatzhalter anstelle konkreter Typen eingefügt wurden, sowie sogenannte ResultSets, die die entsprechenden Typauflösungen liefern. Generische Typen sind ebenfalls Bestandteil des Transferobjekts.
Die Typplatzhalter, die im AST enthalten sind, können durch die konkreten Typinformationen aus dem ResultSet ersetzt werden. Oft gibt es mehrere ResultSets, weil verschiedene gültige Typkombinationen existieren.

```
1 import java.lang.Integer;
2
3 class Test {
4     addOne(i){
5         return i+1;
```

```
    }
}
```

Listing 3. Ursprüngliche Klasse

```
class Test {

Test()({
  }):: TPH V
  TPH N addOne(TPH O i)({
    return (i):: TPH O | 1;
  }):: TPH R

  Test()({
    super(()) Signature: [TPH T];
  }):: TPH U

}
```

Listing 4. Abstrakte Syntax

Ein Beispiel für diese Typplatzhalter ist in Listing 3 und Listing 4 zu finden. Der Code in Listing 3 beinhaltet, wie es in Java-TX üblich ist, keine Typnotationen für die Methode `addOne` und ihren Parameter `i`. Dieser Code wird beim Parsen in den abstrakten Syntaxbaum überführt, wobei die Methode `addOne` den Platzhalter `TPH_N` und der Parameter `i` den Platzhalter `TPH_O` erhält (vgl. Listing 4).
Konkrete Typauflösungen für diese Platzhalter sind im zugehörigen ResultSet (Listing 5) zu finden. Auf diese Weise können die ursprünglichen Platzhalter durch abgeleitete Typen ersetzt werden.

```
[[(TPH P = java.lang.Integer),
  (TPH O = java.lang.Integer),
  (TPH N = java.lang.Integer),
  (TPH Q = java.lang.Integer)]]
```

Listing 5. ResultSet

Im Verlauf des Kompilierungsvorgangs wird auch Bytecode erstellt, der im Verzeichnis `out` abgelegt wird, relativ zum Pfad der kompilierten Datei.
Sollen jedoch nur Syntaxfehler aufgetan werden, muss der Compiler die gesamte Datei immer wieder verarbeiten, um eine vollständige Liste aller Fehler zu erstellen. Das Nachteilige daran ist, dass die Zeitspanne zwischen Anfrage und Antwort sich dadurch erhöht, was für eine Echtzeitverarbeitung in einer IDE problematisch ist. Eine Erweiterung, die dem Entwickler sofortige Rückmeldungen geben soll, kann unter diesen Umständen nur eingeschränkt performant arbeiten.
Das Compiler-Interface bietet deshalb die Möglichkeit, nur nach Parserfehlern zu suchen, ohne die gesamte Typinferenz zu durchlaufen. Hierfür kommt die ANTLR-Grammatik des Java-TX-Compilers zum Einsatz. In diesem Fall gibt das Parser-Interface syntaktische Fehler aus. Eine Übersicht der Interfaces können in Abbildung 11 gesehen werden.

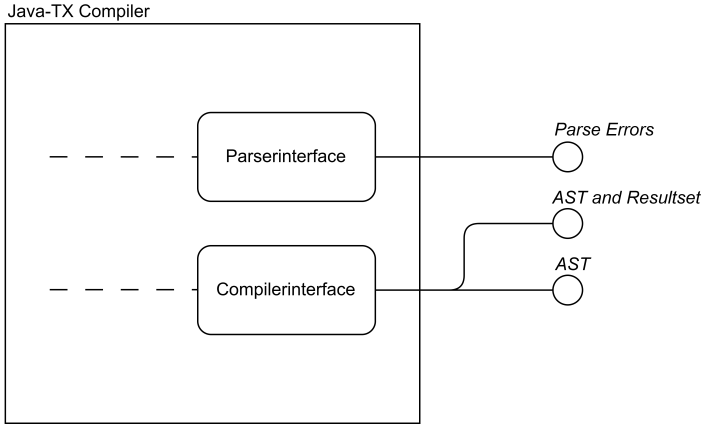

Fig. 11. Die Parserinterfaces

5.1.2 Language Server. Der Language Server nutzt das Compiler-Interface, speichert und aktualisiert interne Datenstrukturen und reagiert auf die Anfragen des Clients. Weil die Kommunikation mit dem Compiler-Interface eine vergleichsweise hohe Latenz hat, wird versucht, direkte Anfragen an dieses Interface weitestgehend zu umgehen.
Es wurden unterschiedliche interne Mechanismen eingeführt, die es ermöglichen, relevante Positionen und ResultSets direkt im Server anzupassen, wenn am Quelltext geändert oder ein bestimmter Typ ausgewählt wird. Dadurch können Compilerzugriffe, die viel Zeit in Anspruch nehmen, umgangen werden.
Der Language Server bereitet die Daten, die er vom Compiler-Interface erhält, auf und speichert sie zwischen. Typauflösungen für die einzelnen Typplatzhalter aus den ResultSets werden als Inlayhint an den Client gesendet. Außerdem werden Informationsdiagnosen erstellt. Quick-Fixes sind auch verfügbar und erlauben es dem Nutzer, einen Typ auszuwählen, um ihn direkt in den Quellcode einzufügen.

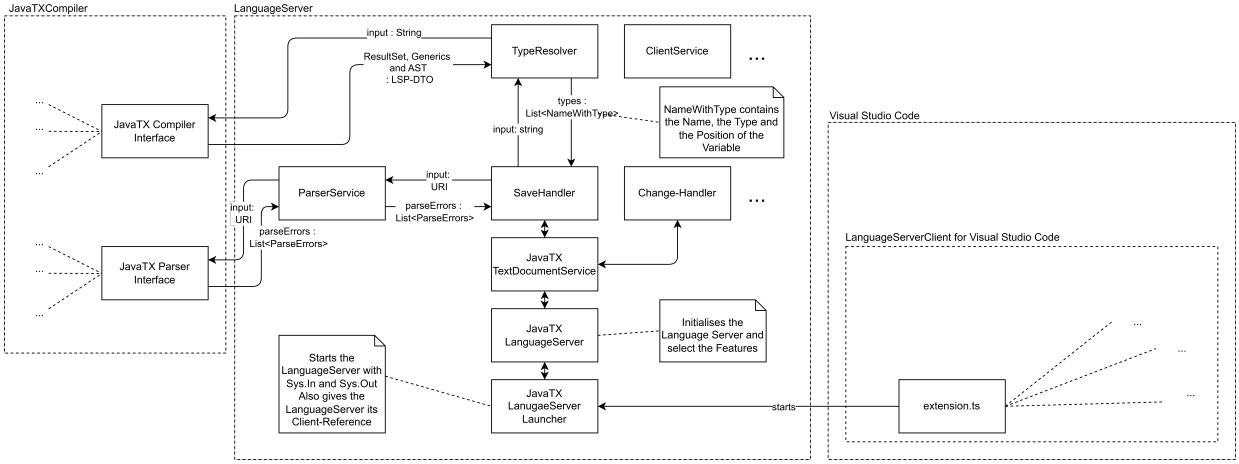

Fig. 12. Genaue Architektur des Langauge Servers. (Nur ein Handler mit Service wird gezeigt)

Die Informationen, die zur Berechnung der Position benötigt werden, stammen aus den Tokenpositionen, die ANTLR generiert und die im AST enthalten sind.
Parserfehler, welche über das Compiler-Interface bereitgestellt werden, sind Diagnosen, die an

den Client zurückgegeben werden. Ein genaues Abbild der Architektur wird in Abbildung 12 dargestellt.

Die Klasse `JavaTXTextDocumentService` ist der zentrale Einstiegspunkt für alle eingehenden Anfragen. Sie implementiert ein Interface, das von LSP4J bereitgestellt wird, und bietet für jede Methode, die in der LSP-Spezifikation definiert ist, eine entsprechende Java-Methode an.
Weil Java-TX nicht alle Funktionen im Protokoll benötigt, haben viele dieser Endpunkte nur leere Rückgaben. Für die Funktionalität von Java-TX sind die folgenden Methoden jedoch unerlässlich:

(1) `change`: Wird ausgelöst, wenn am aktuell geöffneten Dokument Änderungen vorgenommen werden. Parserfehler werden hierbei aktualisiert.
(2) `codeAction`: Ist dafür da, um Quick-Fixes, also Vorschläge für einzufügende Typen, bereitzustellen.
(3) `formatting`: Wird angefordert, wenn eine Dokumentenformatierung gewünscht ist. Dieses Merkmal hat jedoch eine geringere Priorität.
(4) `inlayHint`: Fordert Inlay-Hints an. Dieses Feature ist zusammen mit der Typauswahl das Herzstück der Extension. Sie zeigen inferierte Typen an ohne den Quellcode zu verändern.
(5) `save`: Wenn das Dokument gespeichert wird, wird das Compiler-Interface erneut aufgerufen. Die Cache-Daten werden überschrieben, um einen aktuellen und konsistenten Datenstand zu schaffen.

Handlerklassen, die speziell für die verschiedenen Methoden des Language Servers entwickelt wurden, werden eingesetzt, um Flexibilität zu gewährleisten. Diese Handler übernehmen die Fachlogik und besitzen `handle`-Methoden, die die vom Client gesendeten Parameter annehmen und passende Antworten zurück geben.

(1) `ChangeHandler`: Kümmert sich um alle Änderungen im Dokument. Er ruft das Parser-Interface des Compilers auf und sendet resultierende Diagnostiken direkt an den Client. Die internen Cache-Dateien werden angepasst, um dies zu berücksichtigen. Falls nötig werden die ResultSets eingeschränkt, und illegale ResultSets entfernt. Wenn ein Typ ausgewählt wird, wird einer Variable dann einen festen Wert zugeschrieben. Dadurch können vorher legale ResultSets illegal werden, da die Variable dort einen anderen Typ annimmt. Der entsprechende Typhint, Quickfix und die dazugehörige Diagnostik entfallen. Ohne einen erneuten Kompilierungsvorgang wird der AST abgefragt. Die Typplatzhalter bleiben dabei konsistent, was eine Positionsaktualisierung der Typvariablen ermöglicht. Es erfolgt keine automatische Erkennung neuer Typen. Dies geschieht nur durch eine vollständige Kompilierung, die beim Speichern ausgelöst wird. Alle aktualisierten Typhinweise und Diagnostiken werden erneut gecacht und veröffentlicht.
(2) `CodeActionHandler`: Verantwortlich für das Anbieten der Quick-Fixes. Die Auswahl der Typen erfolgt anhand der gecachten Diagnosen und Typhinweisen. Sobald man einen Typ auswählt, wird über `change` eine neue Anfrage gesendet, die zu einer Positionsaktualisierung und einer Einschränkung der Typauswahl führt.
(3) `FormattingHandler`: Diese Komponente kontrolliert das Dokument auf trailing whitespaces und beseitigt sie, wenn eine Formatierungsanfrage gestellt wird. Es wurden keine weiteren Funktionen umgesetzt.
(4) `SaveHandler`: Eine erneute Kompilierung wird durch das Compiler-Interface angestoßen. Der Cache aktualisiert sich, und Typhints sowie Diagnostiken werden

neu erstellt. Im Gegensatz zum `ChangeHandler` werden hier keine Einschränkungen des ResultSets vorgenommen, weil immer die neuesten Informationen vom Compiler abgerufen werden.

Diese Handler nutzen spezialisierte Services und Hilfsklassen, die nach dem Single Responsibility Principle entworfen sind. Die Anwendungslogik ist dort implementiert, während die Handler die Orchestrierung übernehmen. Die interne Datenstruktur `LSPVariable` wird genutzt, um die enge Beziehung zwischen Typdiagnostiken, Typhints und Quickfixes abzubilden. Sie beinhaltet die Variable, ihren Standort, alle möglichen Typen sowie den ursprünglichen Typplatzhalter. Die entsprechenden Typhinweise und Diagnostiken werden ebenfalls gecacht.

(1) `CacheService`: Zentrale Verwaltung aller temporären Cache-Daten. Setzt sich überwiegend aus Speicherfeldern und passenden Getter-/Setter-Methoden zusammen.
(2) `ClientService`: Service zur Kommunikation mit dem Client. Er verwaltet die Veröffentlichung von Diagnostiken und Benachrichtigungen.
(3) `LogService`: Zuständig für das Logging auf der Client- und Serverseite. Verwendet Log4J und die von LSP4J festgelegten Log-Level.
(4) `ParserService`: Dient zur Kommunikation mit dem Parser-Interface des Compilers. Parserfehler werden durch ihn in Diagnosen umgewandelt.
(5) `TextDocumentService`: Kümmert sich um geöffnete Dateien, speichert Pfadinformationen, erkennt Textänderungen und setzt Vergleichs- sowie Positionslogik für Strings um.
(6) `TypeResolver`: Die zentrale Komponente des Language Servers. Er kommuniziert über das Compiler-Interface und bereitet ResultSets auf und verwaltet diese. Erstellt Inlayhints, Diagnosen und Quick-Fixes. Identifiziert Positionen über Tokeninformationen im AST. Verwendet `GenericHelper` für die getrennte Verarbeitung generischer Typen.
(7) `TypeUtils`: Erstellt konkrete Type-Objekte aus den ResultSets. Gibt alle gültigen Typen für einen spezifischen Typplatzhalter zurück.
(8) `TextHelper`: Abstraktion für spezifische Textoperationen, wie zum Beispiel das Festlegen von Endpositionen in einer Textdatei.
(9) `GenericUtils`: Behandelt generischen Typen, die über ein separates ResultSet bereitgestellt werden.
(10) `DuplicationUtils`: Entfernt redundante Typinformationen damit keine doppelten Typhints und Typauswahllisten bei einer Variable verwendet werden.
(11) `ConversionHelper`: Wandelt interne Modellklassen in Inlayhints und Diagnosen um.
(12) `ASTTransformationHelper`: Transformiert DEN AST in interne Modellklassen, wie beispielsweise Methodenparameter oder Konstruktorvariablen.

Intern werden eigene Modellklassen verwendet, um die Daten strukturiert zu speichern. In der Klasse `LSPVariable` sind unter anderem der Name der Variable, mögliche Typen, Positionen und der ursprüngliche Platzhalter enthalten. Die Klasse `Type` beinhaltet den Namen des Typs und ein Flag, das generische Typen kennzeichnet.
Der gesamte Verarbeitungsfluss startet im `JavaTXTextDocumentService`, das LSP4J-Endpunkte bereitstellt und eingehende Anfragen an die zuständigen Handler weiterleitet. Die Kommunikation zwischen Services und Helfern wird von den Handlern orchestriert. Das Ergebnis wird von den Handler zurückgegeben und an den Client gesendet, möglicherweise begleitet von Fehlern oder Warnungen, etwa bei nicht importierten Typen.
Es gibt neben der fachlichen Logik auch Konfigurationsklassen, die dem Client sagen, welche

Features verfügbar sind. Die Klassen, die LSP4J-Interfaces umsetzen, bieten eine granulare Kontrolle über die Funktionsvielfalt.
Zum Start des Language-Servers existiert eine Klasse mit `main`-Methode, die den Language Server startet.

5.1.3 Editor-Clients. Clients, die mit dem Language Server kommunizieren, sind normalerweise sehr leichtgewichtig und lassen sich durch bereits vorhandene Frameworks umsetzen. Die grundlegende Client-Unterstützung ist in vielen Entwicklungsumgebungen bereits vorhanden, weil die Spezifikation der LSP berücksichtigt wurde. Die Hauptaufgabe eines Clients ist es, den Language Server zu starten, Anfragen zu formulieren und die erhaltenen Informationen richtig in der Benutzeroberfläche darzustellen.
Die Implementierung wird exemplarisch dargestellt, indem sie in Visual Studio Code integriert wird.
Die Erweiterung nutzt die Standardprojektstruktur für Extensions, die Microsoft für Visual Studio Code vorgesehen hat. Dabei wird Typescript verwendet. Node.js wird zum bauen und testen benutzt. Die Bibliothek `vscode-languageclient` wird verwendet, um die LSP spezifischen Funktionen zu bieten.
Die Datei `extension.ts` enthält die Hauptlogik der Extension. Diese Datei legt fest, wann und auf welche Weise der Language Server gestartet wird. Wie in Listing 6 dargestellt, wird der Language Server für alle Dateien mit der Endung `.jav` durch das Pattern `**/*.jav` aktiviert. Danach wird der Server mit festgelegten Optionen initialisiert und gestartet.

```
1  const clientOptions: LanguageClientOptions = {
2          documentSelector: [{ scheme: 'file', language: 'java' }],
3          synchronize: {
4                  fileEvents: vscode.workspace.createFileSystemWatcher(
                       '**/*.jav')
5          },
6          revealOutputChannelOn: 4
7  };
8
9
10
11 const client = new LanguageClient(
12         'javaTxLanguageServer',       // ID des Clients
13         'Java-TX␣Language␣Server',     // Name des Clients
14         serverOptions,
15         clientOptions
16 );
```

Listing 6. Initialisierung des Servers

Im Zentrum der Startlogik steht die Festlegung der `serverOptions` (siehe Listing 7). An dieser Stelle wird bestimmt, wie der Language Server gestartet werden soll. Hierbei wird ein Java-Prozess mit dem Befehl `java -jar ...` gestartet, weil der Language Server über LSP4J in Java umgesetzt ist. Es ist möglich, zwischen Debug- und Normalmodus zu unterscheiden. In diesem Fall gibt es keinen separaten Debug-Modus, weshalb beide Modi denselben Startbefehl nutzen.

```
1  const serverOptions: ServerOptions = {
2          run: {
3                  command: 'java',
```

```
4                   args: ['-jar', workspaceFolder + '/
                        JavaTXLanguageServer-1.0-SNAPSHOT-jar-with-
                        dependencies.jar'],
5           },
6           debug: {
7                   command: 'java',
8                   args: ['-jar', workspaceFolder + '/
                        JavaTXLanguageServer-1.0-SNAPSHOT-jar-with-
                        dependencies.jar'],
9           }
10  };
```

Listing 7. Initialisierung des Servers

Weitere Informationen wie der Name, die Beschreibung oder Symbole der Extension werden in der Datei `package.json` verwaltet. Das Tool `vsce`, welches von Visual Studio Code bereitgestellt wird, wird genutzt, um den finalen Build der Extension zu erstellen. Dies erstellt eine `.vsix`-Datei, die als installierbare Extension fungiert.
Es ist notwendig, dass Java bereits installiert ist, um die Extension auszuführen, da der Language Server auf einer Java-basierten Architektur aufbaut. Weil der Compiler, auf dem alles basiert, ebenfalls in Java geschrieben ist, entsteht dadurch keine zusätzliche Abhängigkeit.

Die Einbindung in andere Entwicklungsumgebungen wie IntelliJ IDEA oder Emacs geschieht auf ähnliche Weise. Für IntelliJ IDEA wird unter anderem das von Red Hat entwickelte Framework `LSP4IJ` eingesetzt. Es ermöglicht die einfache Anbindung eines Language Servers, indem man die Startbefehle und die unterstützten Dateitypen konfiguriert.
In Emacs kann man ebenfalls über die `lsp-mode` eine Integration vornehmen, indem man dort ein Startkommando für den Language Server festlegt. Dies ist zum jetzigen Zeitpunkt jedoch noch nicht umgesetzt. Die `.jar`-Datei des Servers ist unabhängig nutzbar und funktioniert auch ohne eine spezifische IDE-Integration.
Generische LSP-Clients, die über ein Benutzerinterface konfiguriert werden können, existieren neben individuellen Integrationen. Sie erlauben es, beliebige Language Server zu nutzen, indem man ein Startkommando und die zugehörigen Dateimuster angibt. Dies wird zum Beispiel für Intellij IDEA verwendet.

5.2 Kommunikationswege und Datenfluss

Ein vereinfachtes Beispiel zeigt den grundlegenden Datenfluss des Systems. Der Kommunikationsfluss zwischen Client, Language Server und Compilerinterface kann man grundsätzlich in zwei Richtungen unterteilen: Entweder der Client stellt gezielte Anfragen an den Language Server, oder dieser sendet proaktiv Informationen an den Client.
Ein typischer Ablauf startet damit, dass der Nutzer eine Datei speichert. Hierbei schickt der Client eine `save`-Nachricht an den Language Server. Daraufhin ruft der Server das Compilerinterface auf, welches die modifizierte Datei untersucht und entsprechende Ergebnisse (AST, ResultSets, Parserfehler usw.) zurückliefert.
Der Language Server kümmert sich dann um die Verarbeitung der empfangenen Daten. Parserfehler werden als Diagnostiken unmittelbar an den Client gesendet. Währenddessen werden Typhinweise und Quick-Fix-Vorschläge basierend auf den ResultSets erstellt und im internen Cache abgelegt.

Der Client fordert dann weitere Informationen vom Server an, zum Beispiel durch gezielte Abfragen nach InlayHints oder CodeActions. Diese Informationen werden nicht erneut berechnet, sondern direkt aus dem Cache zurückgegeben, weil die Berechnung bereits im Rahmen des vorherigen `save`-Events stattfand.
Darüber hinaus informiert der Language Server den Client über neue Typhinweise und Diagnostiken, was es ermöglicht, dass die Benutzeroberfläche sofort angepasst wird.

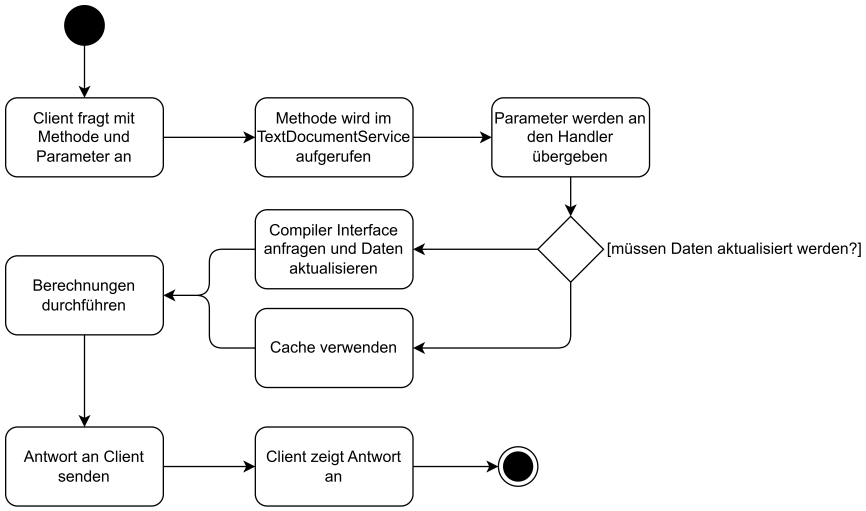

Fig. 13. Basiskommunikationsfluss

Das Kommunikationsmuster, das hier beschrieben und in Abbildung 13 aufgezeigt wird, ist ein typisches Beispiel für den Ablauf zwischen Client und Server, der bei allen Interaktionen ähnlich ist. Egal, welche Methode (z.B. `change`, `hover`, `codeAction`) genutzt wird, die Interaktion läuft immer nach dem gleichen Prinzip ab: Eine Anfrage, eventuell eine Kommunikation mit dem Compilerinterface, die Verarbeitung der Daten und schließlich die Rückgabe an den Client oder ein Push von Informationen durch den Server.

6 Implementierung mit LSP4J und Vorraussetzungen

6.1 Technische Hintergründe

Mindestens Java 23 muss installiert sein, um die Java-TX Language Server Extension auszuführen. Es gibt keine weiteren spezifischen Systemvoraussetzungen. Um den Java-TX-Compiler auszuführen, wird ebenfalls die gleiche Java-Version benötigt, sodass keine weiteren Abhängigkeiten erforderlich sind.
`Maven` wird Intern verwendet, um benötigte Bibliotheken und externe Abhängigkeiten automatisch zu finden und bereitzustellen. Die Struktur des Projekts ist die eines klassischen Maven-Projekts, was eine einfache Integration und Wartung ermöglicht.

Weil während der Typinferenz und Kompilierung viele Berechnungen angestellt werden, kann es bei Programmen, die sehr groß oder komplex strukturiert sind, zu einer spürbaren Verzögerung der Antwortzeit des Language Servers kommen. Eine Caching-Strategie wurde implementiert, um diesem Umstand entgegenzuwirken, indem sie wiederholte Anfragen möglichst performant bedient und die Anzahl der vollständigen Kompilierungsvorgänge minimiert.

7 Integration in Entwicklungsumgebungen

7.1 Visual Studio Code

Die `.vsix`-Datei kann manuell in Visual Studio Code installiert werden.

Die Installation wird in diesen Schritten durchgeführt:

(1) Visual Studio Code starten und zum Erweiterungen-Tab navigieren. Dies ist durch die Suchleiste, die linke Menüleiste oder `Strg + Shift + X` möglich.
(2) Das Drei-Punkte-Menü (...) in der oberen rechten Ecke öffnen.
(3) "Install from VSIX..." aus auswählen.
(4) Zum Ordner, in dem die `.vsix`-Datei, gespeichert ist navigieren und die Datei auswählen. Die Installation geschieht danach automatisch.

Nach der Installation kann die Erweiterung für alle Dateien mit der Endung `.jav` verwendet werden. Bei etwaigen Problemen kann ein Neustart von Visual Studio Code helfen.

7.2 IntelliJ IDEA

Die Integration der Java-TX Language Server Erweiterung in IntelliJ IDEA erfolgt, indem die `.jar`-Datei des Language Servers heruntergeladen und sie dann manuell registriert wird. Anhand diesen Schritten kann die Erweiterung installiert werden.

Zu den Einstellungen unter File > Settings > Plugins navigieren und das Plugin LSP Support (LSP4IJ) von Red Hat suchen und installieren. Mit diesem Plugin können beliebige Language Server über das LSP-Protokoll eingebunden werden.

(1) Nach der Installation kann ein neuer Eintrag mit dem Namen Language Servers in der linken Seitenleiste geöffnet werden.
(2) Einen neuen Eintrag hinzufügen, indem mit der rechten Maustaste auf die Liste geklickt wird und New Language Server ausgewählt wird.
(3) Einen beliebigen Namen verwenden, zum Beispiel `Java-TX`. Im Command-Feld, welches den Befehl enthalten muss um den Language Server zu starten, kann folgender Befehl verwendet werden: `java -jar "<Pfad/zur/LanguageServer.jar>"`
(4) Zum Tab *Maps* wechseln.
(5) Den Punkt *File name* pattern auswählen.
(6) Die Konfiguration um ein neues Mapping erweitern, um den Language Server für Dateien mit der Endung .jav zu aktivieren. Dazu kann auf das + gedrückt und Folgendes hinzugefügt werden: File name pattern: *.jav, Language ID: java_tx
(7) Um die Erweiterung zu verwenden, kann eine `.jav`-Datei geöffnet werden.

Sobald eine `.jav` Datei geöffnet wird, wird unten rechts in der Statusleiste der aktuelle Verbindungsstatus des Language Servers angezeigt.

8 Fazit und Ausblick

8.1 Zusammenfassung der Ergebnisse

Insgesamt wurde ein Language Server erstellt, der die Hauptvorteile von Java-TX aufgreift und gezielt unterstützt. Die Erweiterung, die entwickelt wurde, erlaubt es Entwicklerinnen und Entwicklern, die typlose Syntax von Java-TX zu verwenden, ohne die gewohnte Unterstützung durch moderne Entwicklungsumgebungen aufgeben zu müssen.
Obwohl es noch Einschränkungen gibt, vor allem bezüglich der Laufzeit, umfasst der Language Server bereits eine Reihe von nützlichen Funktionen. Die aufgrund der langsamen Kompilierungsvorgänge des Java-TX-Compilers erforderliche Caching-Strategie kann in einigen Fällen zu inkonsistenten Anzeigen führen. Hier sollten Verbesserungen erfolgen.

Dennoch ist es ein großer Vorteil, dass wichtige Funktionen wie die direkte Kompilierung, die visuelle Darstellung von inferierten Typen und die Auswahl möglicher Typoptionen integriert wurden. Dies ermöglicht es, Feedback zur Typinferenz zu erhalten und mit den bereits berechneten Typen zu arbeiten. Auch Syntaxfehler der Sprache Java-TX werden erkannt und visualisiert, was die Entwicklung vereinfacht und beschleunigt.
Die Erweiterung vereinfacht den Einstieg mit Java-TX: Sie ermöglicht die Entwicklung, ohne einen Konsolen-Workflow nutzen zu müssen, und macht Java-TX dadurch auch für weniger technikaffine Menschen zugänglich.
Es ist besonders bemerkenswert, dass zwei scheinbar gegensätzliche Paradigmen miteinander verbunden werden: Einerseits bewahrt die Java-TX den typlosen Ansatz, während andererseits durch das Berechnen und Anzeigen der Typen die Typsicherheit von Java gewahrt bleibt. So bringt der Language Server die Vorzüge von dynamisch typisierten Sprachen und die Sicherheit von statischen Typisierungen zusammen.

8.2 Nutzen des Language Servers

Die entwickelte Erweiterung hat den entscheidenden Vorteil, dass sie technische Komplexität abstrahiert und den Entwicklungsprozess für Nutzerinnen und Nutzer von Java-TX erheblich vereinfacht. Der Entwicklungsworkflow wird durch den Wegfall der manuellen Kompilierungsschritte über die Konsole und der fehlenden Transparenz über inferierte Typen erheblich verbessert. Java-TX als Programmiersprache wird dadurch insgesamt praktikabler und leichter zugänglich. Vor allem für Entwicklerinnen und Entwickler, die moderne IDEs nutzen, ist die Arbeit mit Java-TX deutlich einfacher, das ist ein wichtiger Faktor für die Nutzbarkeit und die Akzeptanz der Sprache.

8.3 Geplante Erweiterungen

Zukünftige Erweiterungen könnten insbesondere die Unterstützung weiterer Entwicklungsumgebungen umfassen. Die Zielumgebung Emacs, die bereits in den Anforderungen festgelegt wurde, ist noch nicht implementiert.
Durch zukünftige Optimierungen der Compiler-Performance könnte die momentan verwendete cache-basierte Verarbeitung im Language Server überarbeitet und auf eine direkte Kommunikation mit dem Compiler gesetzt werden. Auf diese Weise würden bestehende Ungenauigkeiten oder falsche Darstellungen minimiert.
Hauptfokus liegt jedoch auf den Fehlerkorrekturen und allgemeine Verbesserungen des Language Servers, um die Stabilität und Funktionalität der Erweiterung weiter zu verbessern und bestehende Bugs zu beheben.

References

[1] Eclipse Foundation AISBL. 2016. Eclipse LSP4J | projects.eclipse.org. https://projects.eclipse.org/projects/technology.lsp4j.

[2] JetBrains s.r.o. [n. d.]. IntelliJ IDEA – the IDE for Pro Java and Kotlin Development. https://lp.jetbrains.com/intellij-idea-promo.

[3] Microsoft Corporation. 2025. Official page for Language Server Protocol. https://microsoft.github.io/language-server-protocol.

[4] Martin Plümicke. 2024. Completing the Functional Approach in Object-Oriented Languages. In *A Second Soul: Celebrating the Many Languages of Programming - Festschrift in Honor of Peter Thiemann's Sixtieth Birthday,* Freiburg, Germany, 30th August 2024, Annette Bieniusa, Markus Degen, and Stefan Wehr (Eds.). Electronic Proceedings in Theoretical Computer Science, Vol. 413. Open Publishing Association, 43–56. doi:10.4204/EPTCS.413.4

Umsetzbare Abbildung von Untersorten und partiellen Operationen auf Many-Sorted-Algebra mit Konstruktoren

EDWARD SABINUS, Martin-Luther-Universität Halle-Wittenberg, Nat. Fak. III, Deutschland

Untersorten und partielle Operationen sind grundlegende Konzepte der Programmierung und doch werden sie in Sprachen wie Haskell, Coq oder Isabelle/HOL nicht unterstützt. Die Lösung dafür ist eine Abbildung von Order-Sorted-Algebra (OSA) auf die Many-Sorted-Algebra (MSA) unter Berücksichtigung von Konstruktoren. Sowohl Common-Algebraic-Specification-Langauge (CASL) als auch OSA geben eine Abbildung von Untersorten auf die MSA vor. OSA diskutiert auch Ansätze zur Darstellung von partiellen Operationen. Jedoch berücksichtigt die Spezifikation dieser Abbildung die Konstruktoren nicht, sodass die Umsetzung dieser Abbildung in eine der oben genannten Sprachen nicht möglich ist. In diesem Papier werden die Grenzen der Abbildung aus CASL und OSA auf MSA in Bezug auf die Umsetzbarkeit mit Konstruktoren gezeigt und eine umsetzbare Abbildung vorgestellt, die die Konstruktoren berücksichtigt.

Author's Contact Information: Edward Sabinus, Martin-Luther-Universität Halle-Wittenberg, Nat. Fak. III, Halle, Deutschland, edward.sabinus@informatik.uni-halle.de.

Enhancing Security and Robustness in Cyber-physical Systems with the Lemming Runtime System

NILS SCHEIDWEILER, Friedrich-Schiller-Universität Jena, Deutschland

For cyber-physical systems (CPSs) non-functional properties such as energy, time, security, and robustness (ETSR) are as important as functional correctness. The deployment of CPSs on heterogeneous and parallel computing platforms faces complex challenges such as solving scheduling and mapping problems to meet ETSR constraints and objectives.

We propose Lemming, our novel runtime system to tackle these challenges. It adopts the TeamPlay coordination model: applications are organized as directed acyclic graphs (DAGs). Tasks (vertices) implement application-specific code and channels (edges) define both task dependencies and stream-based data exchange.

Lemming serves as a highly configurable toolbox, making it adaptable to many real-world applications with varying ETSR properties. We present Lemming's general design and then focus on its security and robustness features. They include process-based task isolation, containment of tasks, as well as a custom kernel-space process scheduler, which is designed with a strong emphasis on safety considerations.

Author's Contact Information: Nils Scheidweiler, Friedrich-Schiller-Universität Jena, Jena, Deutschland, nils.scheidweiler@uni-jena.de.

A Brief Comparison of Module Systems in SML and Java

JULIAN SCHMIDT, Baden-Wuerttemberg Cooperative State University, Germany

Since its release, SML has inspired many functional programming languages—such as Haskell, OCaml or F#—with features like type inference, algebraic data types (ADTs) and higher order functions. Over the past decades, many of these features have also been—at least partially—adopted to object-oriented languages, such as Java. Another powerful feature of SML is its module system. However, this aspect of SML has not been fully embraced in OOP languages, which typically rely on classes and interfaces to group related code. Java has introduced a module system in Version 9, called the JPMS. This document gives a small introduction to the Java and SML module systems and briefly compares the two languages with regards to their abstraction capabilities.

1 SML's Module System

Standard ML's (SML's) module system is infamous among functional programming languages. It encourages modularization of cohesive code into modules and therefore promotes code reuse. The module system of SML is distinct from the core language [3] and consists of the following three concepts:

- Structures
- Signatures
- Functors

In the following sections, an overview of these concepts is provided.

1.1 Structures

A SML structure packages related elements such as functions and values into a module. That means multiple related elements are combined into a single container.

```
structure Stack =
  struct
    exception E;
    val empty = [];
    fun push(q,x) = ...
    fun peek(x::q) = ...
    fun pop(x::q) = ...
    fun size(x::xs) = ...
  end;
```

Listing 1. An example stack structure in SML

Listing 1 shows an example of a Stack structure implemented using a list. As the implementations of the functions are mostly straigtforward and add little to the discussion, they are omitted for brevity in this and the following examples. In SML, the `struct - end` block is used to define a structure [6][p. 60]. In between this block is the actual definition of exceptions, values and functions contained in the structure. As the stack is implemented based on a list, we denote the empty stack to be the empty list and implement the common stack methods by pattern matching on the builtin list data type. Notice that we don't need to provide any type information, as SML can infer all types during compile time. This block is then assigned to a structure named `Stack`.
The structure can then be used in the REPL as follows:

Author's Contact Information: Julian Schmidt, Baden-Wuerttemberg Cooperative State University, Horb, Germany, j.schmidt@hb.dhbw-stuttgart.de.

```
- val s = Stack.empty;;
val s = [] : 'a list
- val s1 = Stack.push(s, 42);;
val s1 = [42] : int list
- Stack.peek(s1);;
val it = 42 : int
```

First `Stack.empty` is assigned to a new variable s. This assigns an empty list of arbitrary elements to the variable, as this is how we defined the empty value in Listing 1. Next the value 42 is pushed onto the empty Stack. At this point it is evident, that the list's elements need to be of type `int`, which is also returned by the Read–eval–print loop (REPL). Lastly we use the peek function to determine the top element of the stack, which the REPL correctly evaluates to 42. As we didn't assign the result to a new variable, the REPL automatically assigned the value to the variable `it`. One caveat of this approach is, that the internal data type of the stack is not opaque. That is, the list type is exposed to the outside and can be used with the stack functions directly, as the following example shows:

```
- Stack.push([2,3], 1);;
val it = [1,2,3] : int list
```

We will come back to this in the next section, to show how we can hide the internal data type of a structure.

1.2 Signatures

Signatures define a contract, which each structure that matches the signature, must satisfy. That is, a signature abstractly defines specifications, a structure must implement.
Listing 2 shows an example signature for the stack module.

```
signature STACK =
  sig
    type 'a t;
    exception E;
    val empty : 'a t;
    val push : 'a t * 'a -> 'a t;
    val peek : 'a t -> 'a;
    val pop : 'a t -> 'a t;
    val size : 'a t -> int;
  end;
```

Listing 2. An example stack signature in SML

At the beginning, we denote a new abstract type `'a t`, which we use throughout the function signatures. In SML `'a` is a type variable, i.e. it stands for an arbitrary type [6][p. 65]. For now `t` is an abstract type, which means we leave it to the structure to define the type. This also means multiple structures which match the signature, can have different implementations of the type. For the stack functions, we omitted all implementation details of the functions and instead provided the types. In SML, the asterisk (*) denotes the Cartesian product, which represents tuples. So `A * B` is the type of the tuple `(a: A, b: B)`
Since the structure in Listing 1 matches the signature in Listing 2, we can explicitly ascribe the structure to the signature.

```
structure Stack1 : STACK = Stack;
```

This constrains the structure to satisfy the signature [6][p. 269]. That is, if the structure does not implement every function and specify each abstract type, the code won't compile.
However with this approach, the internal data type of the Stack (i.e. 'a list) is still exposed. We can prevent this from using the opaque ascription :> instead of the transparent ascription : [6][p. 269-270].

```
structure Stack1 :> STACK = Stack;
```

With this change, the internal types of the structure are now hidden and only the abstract types of the signature, are present to the user of the module. The example from above will result in an error:

```
- Stack1.push([2,3], 1);;
Error: operator and operand do not agree
```

1.3 Functors

Functors are functions on modules and are not to be confused with Haskell functors. Functors in SML are completely separate from the core language and take modules as input to return a new module. This comes in handy in a number of scenarios, but let's look at a simple example to clarify this concept.
Suppose we want to create a module which represents an ordered set. We could yet again use the builtin list data type to represent a list, but we still run into a problem: We cannot allow the set to hold any data type. Since we want to create ordered set, we need to make sure, that an order is defined on the items in the set. Basically we want to constraint the types of elements which the set can hold to data types that can be ordered.
To achieve this, we start by generating a new signature which defines function, that compares elements of an arbitrary type `t`.

```
datatype order = LT | EQ | GT
signature ORDERED =
  sig
    type t;
    val compare t * t -> order;
  end;
```

With that, we can now write a functor that takes a structure which matches our `ORDERED` signature and generates a new structure that represents a sorted set. For the comparison of the elements, we use the compare function from the `ORDERED` signature.

```
functor GenOrderedSet (O : ORDERED) =
  struct
    exception E
    type elem = O.t
    type set = elem list

    val empty = []
    fun add([], x) = ...
    fun remove([], x) = ...
    fun contains([], x) = ...
  end;
```

Yet again, this use of a functor can be thought of as a constraint, i.e. the functor will provide an ordered set of elements type `t`, iff one explicitly provides an order for type `t`. This is similar

to Haskell type classes, with the key difference, that Haskell automatically passes the correct implementation for a type while the module needs to be passed explicitly in SML.

2 Java's module system

With version 9, the Java Platform Module System (JPMS) was introduced with two primary goals in mind [5].

- Reliable configuration, to replace the brittle, error-prone class-path mechanism with a means for program components to declare explicit dependences upon one another.
- Strong encapsulation, to allow a component to declare which of its public types are accessible to other components, and which are not.

This is done by extending the typical package-based modularization approach—which is primarily used for namespacing—through so-called modules. Similar to how multiple classes or interfaces can be bundled into packages, Java 9 provides the possibility to bundle multiple packages—and other data—into modules [2]. However, we will see that the definition of modules in Java is quite different from modules in SML.
The big new addition is a `module-info.java` file, which is located at the root of the projects package hierarchy and contains a module declaration [2, Section 1.1]. With it, we specify the name of our module, along with additional information to ensure **reliable configuration** and **strong encapsulation**.
Furthermore the module path replaces the class path for modules, i.e. module locations are provided in the module path as opposed to the class path [2, Section 2.1]. To ensure backwards compatibility, however, the class path is still available.
A basic example of a module declaration is given in Listing 3. In this case, we only define the name of the module. In the next two sections, the content will be extended with additional information.

```
module mymodule {}
```

Listing 3. "Basic declaration of a module-info.java file"

2.1 Reliable Configuration

Before Java 9 there was no way to declare explicit dependencies between code, directly in the Java language. Java would just search the classpath and try to find all the imports. One would therefore place all their dependencies onto the classpath and compile the project. This is typically done by build assistants like gradle or maven.
With the addition of the module system however, it is not only possible, but necessary to declare explicit dependencies [2]. This works only at the module level, i.e. a module declares which other modules it is required to compile and run. With the release of the module system, the JDK itself was also modularized [2, Section 1.4]. For example `java.base` contains the base code for the Java ecosystem, such as the packages `java.lang` and `java.util`. It is implicitly required by every other module, because it is mandatory for the JVM. Other JDK modules include `java.sql` for SQL related functionality and `java.desktop` for UI related tasks.
Suppose we want to access a SQL database in our code, so let's add the `java.sql` module to the dependencies in our module declaration file. We declare our dependency at both compile time and run time with the `requires` keyword.

```
module mymodule {
   requires java.sql;
}
```

This will ensure that during both compile time and run time, the module is available on the module path. In other words, the module system is tightly integrated into the java ecosystem with both the compiler and the JVM being aware of it.

2.2 Strong Encapsulation

Strong encapsulation provides a control mechanism to define which packages are for internal use and which packages should be exposed to other modules. That is, we can hide our internal code from outside use at package level.
Let's suppose we have the following structure in our example module:

```
├── module-info.java
└── org/
    └── example/
        ├── base/
        │   └── Converter.java
        └── internal/
            └── Utils.java
```

The `org` package is the root of our package hierarchy. We want to provide the `org.example.base` package for external use but keep the `org.example.internal` package hidden, as this is the location of the internal code. This code might change in the future and should not be used by external code. To denote which packages we want to expose for external use, we explicitly declare them in our `module-info.java` file. This is done with the `exports` keyword. Every package we do not explicitly declare here, can't be used from outside the module [2, Section 1.1]. This is not only ensured by the compiler during compile time, but also by the JVM at run time.
It is also possible to denote compile-time-only dependencies with the `requires static` option, respectively [1, §7.7.1]. Furthermore the module system does provide the possibility to restrict reflection access at runtime and an approach to load services. There are many other keywords and features of the JPMS, which can be read about in the Java Language Specification [1, §7].

```
module mymodule {
    requires java.sql;
    exports org.example.base;
}
```

3 Comparison of Abstraction Capabilities

It is clear that the JPMS and the module system of SML solve different problems.
The JPMS focuses on declaring dependencies and access control, while the SML module system enables modular programming by grouping related definitions into structures, specifying interfaces via signatures, and parameterizing modules through functors, which allows for encapsulation and abstraction [6, p. 313]. In Java, similar functionality is typically achieved using `classes` to group related code, `interfaces` to abstract over implementations and `packages` for namespace management.
These OOP concepts map vaguely to the concepts of the SML module system: For example, one might write the Stack example from Listing 2 in Java as shown in Listing 4. However, there is no direct mapping to SML functors in Java and abstract types seems to correspond only approximately to parametric polymorphism.

```java
import java.util.LinkedList;
interface IStack<T>{
    void push(T x);
    T pop();
    boolean empty();
    int size();
    T peek();
}

class Stack<T> implements IStack<T>{
    private final LinkedList<T> elements =
        new LinkedList<>();

    public void push(T x) {...}
    public T peek() {...}
    public T pop() {...}
    public int size() {...}
}
```

Listing 4. Stack implementation in Java

Compared to the SML implementation, we use an internal state in the `Stack` class, representing the stack as a list. Since the lists are mutable in Java, we do not need to return the data structure after each modification. Instead we mutate the internal list, following a more OOP-oriented approach. Similar to the opaque type abstraction in SML, we can also prohibit access on the internal data structure by declaring the field as `private`. Furthermore, we use generic type parameters, (i.e., `<T>`) to emulate SML's abstract types. Compared to the SML approach, the selection of the type is done at the time of instantiating the `Stack` class.

4 Conclusion

In this document, a brief comparison between Java's and SML's module system was provided. It is evident that, although they share the name "module system", they solve different problems. It seems more accurate to compare Java's OOP concepts like `classes`, `interfaces` and `packages` to SML's module system features like `signatures`, `structures` and `functors`. Nonetheless, these are still distinct abstraction mechanisms and provide different features in some aspects, i.e. there is no direct comparison to functors in Java.
Another interesting aspect would be to evaluate how abstract types and parametric polymorphism (i.e. generics) correlate. For further comparison, it would be valuable to look at Scala, another JVM language. Compared to Java, Scala supports and emphasizes the use of abstract types in addition to generics, seeming to offer a SML-like abstraction [4, Section 5.2].

References

[1] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification, Java SE 24 Edition.* Oracle, 2025.

[2] Mark Reinhold. The state of the module system. https://openjdk.org/projects/jigsaw/spec/sotms/, 2016. Accessed: 2025-08-28.

[3] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[4] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[5] OpenJDK. Project jigsaw. https://openjdk.org/projects/jigsaw/, 2017. Accessed: 2025-08-28.
[6] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

Ein Typsystem für eine deklarative Sprache über ausführbare Zufallsexperimente

BALTASAR TRANCÓN WIDEMANN, TH Brandenburg / semantics gGmbH, Deutschland

Alea ist eine domänenspezifische deklarative Programmiersprache für Zufallsexperimente. Programme können auf zweierlei Arten interpretiert werden; einerseits statisch als Wahrscheinlichkeitsverteilung aller möglichen Ergebnisse, andererseits dynamisch als pseudozufällige Stichprobe. Alea soll als didaktisches Werkzeug in der Stochastik-Lehre dienen, als Simulationsumgebung, sowie in Entwurf, Analyse und Implementierung von Zufallselementen in Spielen. Die Benutzung der Sprache soll zwar elementare Mathematikkentnisse, aber möglichst wenig Programmiererfahrung erfordern. Im Vortrag wird der Entwurf eines Typsystems vorgestellt, dass diesen Anforderungen genügt.

Author's Contact Information: Baltasar Trancón Widemann, TH Brandenburg / semantics gGmbH, Brandenburg / Berlin, Deutschland, trancon@th-brandenburg.de.

Revising the TeamPlay Coordination Language: JenPlay

ERIC WINTZLER, Friedrich-Schiller-Universität Jena, Deutschland

For writing a stream processing program, a specialised programming language like TeamPlay can be used. TeamPlay is a coordination language proposing the idea of separating computation and coordination and describes an streaming application as a directed acyclic graph (DAG) of stateless components.

We present the JenPlay coordination language. JenPlay is both a revision and an extension of the TeamPlay language with the general direction of making the language more expressive and to scale to larger systems. Focus areas of this work-in-progress presentation are the separation between component definition and instantiation, the seamless composition of larger DAGs out of previously defined ones and the structuring of channel lists.

Author's Contact Information: Eric Wintzler, Friedrich-Schiller-Universität Jena, Jena, Deutschland, eric.wintzler@uni-jena.de.