DHBW
Duale Hochschule
Baden-Württemberg
Stuttgart

Engineering

# INSIGHTS

Schriftenreihe der Fakultät Technik: 1/2025

## Tagungsband des Jahrestreffens 2025 der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“

Daniel Holle, Prof. Dr. Jens Knoop, Prof. Dr. habil. Martin Plümicke, Prof. Dr. Peter Thiemann,
Prof. Dr. habil. Baltasar Trancón y Widemann (Hrsg.)

Daniel Holle

Duale Hochschule Baden-Württemberg (DHBW) Stuttgart Campus Horb
Studiengang Informatik
Florianstraße 15
72160 Horb am Neckar

E-Mail: d.holle@hb.dhbw-stuttgart.de

Prof. Dr. Jens Knoop

Technische Universität Wien
Fakultät für Informatik
Argentinierstr. 8
1040 Wien
Österreich

E-Mail: jens.knoop@tuwien.ac.at

Prof. Dr. habil. Martin Plümicke

Duale Hochschule Baden-Württemberg (DHBW) Stuttgart Campus Horb
Studiengang Informatik
Florianstraße 15
72160 Horb am Neckar

E-Mail: m.pluemicke@hb.dhbw-stuttgart.de

Prof. Dr. Peter Thiemann

Universität Freiburg
Institut für Informatik
Georges-Köhler-Allee Geb.079
79110 Freiburg i. Br.

E-Mail: thiemann@informatik.uni-freiburg.de

Prof. Dr. habil. Baltasar Trancón y Widemann

Technische Hochschule Brandenburg
Fachbereich Informatik und Medien
Praktische Informatik
Magdeburger Straße 50
14770 Brandenburg an der Havel

E-Mail: trancon@th-brandenburg.de

Die Teilnehmenden des Workshops

## Vorwort

Seit 1984 veranstaltet die GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“ jedes Frühjahr einen Workshop im Physikzentrum Bad Honnef. Das Arbeitstreffen ist eine wichtige Plattform für Informatikerinnen und Informatiker aus dem deutschsprachigen Raum zum Erfahrungsaustausch, der Diskussion, zum Netzwerken und zur Vertiefung von Kontakten. Der Workshop wird getragen sowohl von den Universitäten, deren Professoren ihn mal ins Leben gerufen haben, als auch von Hochschulen angewandter Wissenschaften und der Dualen Hochschule Baden-Württemberg.

Vorgestellt werden Vorträge und Demonstrationen sowohl zu abgeschlossenen als auch laufenden Arbeiten, darunter Themen wie

- Sprachen, Sprachparadigmen
- Korrektheit von Entwurf und Implementierung
- Werkzeuge
- Software-/Hardware-Architekturen
- Spezifikation, Entwurf
- Validierung, Verifikation
- Implementierung, Integration
- Sicherheit (Safety und Security)
- eingebettete Systeme
- hardware-nahe Programmierung
- Künstliche Intelligenz

In diesem Tagungsband wurden Beiträge des 41. Workshops aufgenommen, der vom 9. bis 11. April 2025 stattfand. Die Autorinnen und Autoren reichten Zusammenfassungen ihrer Beiträge genauso wie komplette Artikel ein. Auf dem Programm standen alle Kernthemen des Gebietes, unter anderem Typen, Compiler, Sprachdesign, Künstliche Intelligenz und maschinennahes Programmieren. Außerdem wurden Prototypen von Werkzeugen demonstriert.

Wir danken allen Teilnehmenden, die den Workshop mit ihren Vorträgen, Abstracts, Papers und lebendigen Diskussionen aufs Neue zu einem interessanten und aufschlussreichen Ereignis machten. Ein besonderer Dank gilt den Mitarbeitenden des Physikzentrums Bad Honnef, die durch ihre umfassende Betreuung für eine angenehme und anregende Atmosphäre gesorgt haben.

Daniel Holle, Jens Knoop, Martin Plümicke,
Peter Thiemann, Baltasar Trancón Widemann
Dezember 2025

## Inhaltsverzeichnis

*Michael Hanus* – Can Logic Programming Be Liberated from Predicates and Backtracking? **3**

*M. Anton Ertl* – Multi-precision integer arithmetics **15**

*Björn Lötters* – Noo: Towards a Meta-Language Calculus (Abstract) **27**

*Daniel Holle* – Pattern Matching in Java-TX **29**

*Nils Scheidweiler & Clemens Grelck* – Lemming: A Novel Runtime System for Cyber-physical Systems **37**

*Andreas Stadelmeier* – Type Inference for Java using ASP – First approach **45**

*Martin Plümicke* – The Java-TX Roadmap **57**

# Can Logic Programming Be Liberated from Predicates and Backtracking?

Michael Hanus
Institut für Informatik, Kiel University, Germany
mh@informatik.uni-kiel.de

## Abstract

Logic programming has a long history. The representative of logic programming in practice, the language Prolog, has been introduced more than 50 years ago. The main features of Prolog are still present today: a Prolog program is a set of predicate definitions executed by resolution steps with a backtracking search strategy. The use of backtracking was justified by efficiency reasons when Prolog was invented. However, its incompleteness destroys the elegant connection of logic programming and the underlying Horn clause logic and causes difficulties to teach logic programming. Moreover, the restriction to predicates hinders an adequate modeling of real world problems, which are often functions from input to output data, and leads to unnecessarily inefficient executions. In this paper we show a way to overcome these problems. By transforming predicates and goals into functions and nested expressions, one can evaluate them with a demand-driven strategy which might reduce the number of computation steps and avoid infinite search spaces. Replacing backtracking by complete search strategies with new implementation techniques closes the gap between the theory and practice of logic programming. In this way, we can keep the ideas of logic programming in future programming systems.

## 1 Introduction

Logic programming was developed as a restriction of the general resolution principle [34] to Horn clauses so that efficient linear (SLD-resolution) proofs can be constructed (see also [14] for some historical background). It became popular when concrete implementations in the form of interpreters (and later compilers) for the programming language Prolog were available. Horn clauses and SLD-resolution are tightly connected to mathematical logic. The soundness and completeness of SLD-resolution establish the foundation of logic programming [28]. Unfortunately, the memory restrictions of computers at that time caused a gap between these theoretical foundations and the practice of logic programming in Prolog: non-deterministic computations are evaluated by backtracking so that the theoretical completeness of SLD-resolution is lost. For instance, consider the definition of a Prolog predicate relating a list and its last element:

```
last([H|T],E) :- last(T,E).
last([E],E).
```

This definition works when the list is known:

```
?- last([1,2,3],E).
E = 3
```

One of the advantages of logic programming is the absence of fixed input and output parameters. Instead of providing a known value for an argument of a predicate, one can also call the predicate with a free variable for this argument (as `E` above) so that a result is computed by binding this variable to some value. In practice, this advantage is often lost when non-deterministic

search is implemented by backtracking, since infinite branches in a search tree might preclude the computation of valid answers. For instance, Prolog does not compute any result for the definition of `last`, as shown above, when the list is unknown, e.g., for the goal `last(L,3)`: the backtracking strategy causes an infinite chain of applications of the first rule. This shows the gap between the theory of logic programming, where the complete SLD-resolution method yields an infinite set of answers to this goal, and the practice of logic programming implemented with the language Prolog.

Compared to functional programming, logic programming is often considered as the more flexible and expressive programming paradigm [33]. This is no longer true if we consider functional logic languages [6], such as Curry [23], which amalgamates features of functional and logic programming and does not force the programmer to model all knowledge in the form of predicates. Actually, many real world problems can be modeled in a more adequate format in the form of functions mapping input data to output data. With a functional logic language, one has the same expressiveness as in logic programming since any (pure) logic program can be transformed into a functional logic program so that the same solutions are computed, as we discuss in this paper. Moreover, the equivalent functional logic programs behave more efficiently and can avoid infinite search spaces.

**Example 1.** Consider the following Prolog program which defines the well-known predicate `app` relating two lists to their concatenation and a predicate `app3` relating three lists to their concatenation:

```
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

app3(Xs,Ys,Zs,Ts) :-
  app(Xs,Ys,Rs), app(Rs,Zs,Ts)
```

When evaluating the goal `app3(Xs,Ys,Zs,[])`, a Prolog system yields the answer

```
{Xs↦[], Ys↦[], Zs↦[]}
```

but it does not terminate when searching for more answers. Similarly, Prolog systems do not terminate when evaluating the goal `app3(Xs,[1],Zs,[])`. This is different if we translate the predicates into functions by considering the last argument as output, as often intended when formulating functional knowledge as predicates. For instance, the tool described in [17] translates these definitions into the following Curry program:

```
app []     ys = ys
app (x:xs) ys = x : app xs ys

app3 xs ys zs = app (app xs ys) zs
```

Since Curry exploits functional dependencies between input and output data to implement a demand-driven strategy, the equations (which are equivalent to the goals above) `app3 xs ys zs =:= []` and `app3 xs [1] zs =:= []` have finite search spaces so that the evaluation in Curry terminates.

In summary, we can see that the basis of Prolog, i.e., predicates and backtracking, has various disadvantages:

- The theoretical completeness of SLD-resolution is lost.
- Backtracking hinders teaching the ideas of logic programming since beginners are often faced with the influence of the search strategy.
- Programmers have to think about the influence of backtracking to the success of computations—a contradiction to the idea of *declarative programming*.
- The use of predicates instead of functions yields a flat structure of goals so that functional dependencies cannot be exploited to avoid useless search.

In this paper we argue that all these problems can be avoided without loosing the flexibility of logic programming by using *functional* logic programming instead of pure logic programming. Functions are helpful to reduce the number of computation steps and avoid infinite search spaces. Contemporary functional logic languages, such as Curry, do not fix a deterministic backtracking strategy for search but support complete search strategies.[1] Thus, abandoning backtracking in logic programming is similar to the removal of the von Neumann bottleneck by functional programming [10]: it supports a higher, declarative programming style which frees the programmer from thinking about low-level control details.

In the following, we sketch[2] methods to get rid of predicates and backtracking. This can be done in

[1] Note that this is not the case for all such languages. For instance, the functional logic language Verse [8] fixes a deterministic, backtracking-like search strategy.

[2] More details can be found in [11, 17] on which this paper is partially based.

a systematic way by transforming logic programs into functional logic programs on which efficient, often optimal, and complete evaluation strategies can be applied. To explain this method, we review the basics of logic and functional logic programming in the next section. Then we show how to transform pure logic programs into functional logic programs and how to apply efficient and complete evaluation strategies on the transformed programs.

The message of this paper is to show that functional logic languages are always preferable to pure logic languages. Transforming logic into functional logic programs is the formal justification. If one accepts this message, one should directly implement the desired application in a functional logic language and exploit all useful features of such languages, like declarative I/O [37], functional patterns [4], strategy-independent encapsulated search [7], default rules [5], etc.

## 2 Logic and Functional Logic Programming

We briefly review some notions and features of logic and functional logic programming. More details can be found in [28] and in surveys on functional logic programming [6, 18].

We use Prolog syntax to present logic programs. *Terms* in logic programs are constructed from variables $(X, Y, \ldots)$, numbers, atom constants $(c, d, \ldots)$, and functors or term constructors $(f, g, \ldots)$ applied to a sequence of terms, like $f(t_1, \ldots, t_n)$. A *literal* $p(t_1, \ldots, t_n)$ is a predicate $p$ applied to a sequence of terms, and a *goal* $L_1, \ldots, L_k$ is a sequence of literals, where $\square$ denotes the empty goal $(k = 0)$. *Clauses* $L$ :- $B$ define predicates, where the *head* $L$ is a literal and the *body* $B$ is a goal (a *fact* is a clause with an empty body $\square$, otherwise it is a *rule*). A *logic program* is a sequence of clauses.

Logic programs are evaluated by SLD-resolution steps, where we consider the leftmost selection rule here. Thus, if $G = L_1, \ldots, L_k$ is a goal and $L$ :- $B$ is a variant of a program clause (with fresh variables) such that there exists a most general unifier[3] (*mgu*) $\sigma$ of $L_1$ and $L$, then

$$G \vdash_\sigma \sigma(B, L_2, \ldots, L_k)$$

is a *resolution* step. A *computed answer* for a goal $G$ is a substitution $\sigma$ (restricted to the variables occurring in $G$) which is composed of all unifiers of a sequence of resolution steps from $G$ to $\square$.

[3] Substitutions, variants, and unifiers are defined as usual [28].

**Example 2.** Consider the predicates of Example 1 and the list reversal

```
rev([],[]).
rev([X|Xs],Zs) :-
  rev(Xs,Ys),
  app(Ys,[X],Zs).
```

The predicate `pali` relates a palindrome with its middle element:

```
pali(Zs,X) :-
  app3(Xs,[X],Ys,Zs),
  rev(Xs,Ys).
```

Prolog computes for the goal

```
pali([1,2,3,2,1],M)
```

the answer $\{\mathtt{M} \mapsto \mathtt{3}\}$ but then it does not terminate, since it enumerates arbitrary large values for `Xs`. Similarly, it does not terminate on `pali([1,2],M)`.

Functional logic programming [6, 18] integrates the most important features of functional and logic languages, such as higher-order functions and lazy (demand-driven) evaluation from functional programming and non-deterministic search and computing with partial information from logic programming. The declarative multi-paradigm language Curry [23], which we use in this paper, is a functional logic language with advanced programming concepts. Its syntax is close to Haskell [31], i.e., variables and names of defined operations start with lowercase letters and the names of data constructors start with an uppercase letter. The application of an operation $f$ to $e$ is denoted by juxtaposition ("$f\ e$").

In addition to Haskell, Curry allows *free* (*logic*) *variables* in program rules (equations) and initial expressions. Function calls with free variables are evaluated by a possibly non-deterministic instantiation of arguments. Similarly to Prolog and in contrast to Haskell, Curry evaluates operations defined by rules with overlapping left-hand sides in a non-deterministic manner by applying all possible rules. The archetype of an operation defined by overlapping rules is the non-deterministic choice, defined in Curry [23] as the infix operator "?" by

```
x ? _ = x
_ ? y = y
```

Hence, "`0 ? 1`" yields two values: `0` and `1`. In contrast to Prolog, the concrete strategy to compute these values, i.e., the search strategy, is not fixed in Curry so that implementations of Curry can provide various search strategies.

**Example 3.** The following Curry program[4] defines the predicates of Example 2 in a functional manner, where logic features (like the free variables `xs` and `x`) are exploited to define `pali`:

```
rev []     = []
rev (x:xs) = app (rev xs) [x]

pali zs | zs =:= app3 xs [x] (rev xs)
        = x
```

"`|`" introduces a condition, and "`=:=`" denotes semantic unification, i.e., the expressions on both sides are evaluated before unifying them.

Since `app` and `app3` can be called with free variables in arguments, the condition in the definition of `pali` is solved by instantiating `xs` and `x` to appropriate *values* (i.e., expressions without defined functions) before reducing a function call. This corresponds to narrowing [32, 35]. $t \leadsto_\sigma t'$ is a *narrowing step* if there is some non-variable position $p$ in $t$, an equation (program rule) $l$ = $r$, and an mgu $\sigma$ of $t|_p$ and $l$ such that $t' = \sigma(t[r]_p)$,[5] i.e., $t'$ is obtained from $t$ by replacing the subterm $t|_p$ by the equation's right-hand side and applying the unifier. Conditional equations $l$ | $c$ = $r$ are considered as syntactic sugar for the unconditional equation $l$ = $c$ &> $r$, where "&>" is defined by `True &> x = x`.

Curry is based on the *needed narrowing strategy* [2] which uses non-most-general unifiers in narrowing steps to ensure the optimality of computations. Needed narrowing is a demand-driven evaluation strategy, i.e., it supports computations with infinite data structures [26] and can avoid superfluous computations so that it is optimal w.r.t. the number of computed solutions and the length of derivation [2]. This is our motivation to transform logic programs into Curry programs, since it can reduce infinite search spaces to finite ones. For instance, the evaluation of the expression `pali []` has a finite computation space: the generation of larger lists for the first argument of `app3` is avoided since there is no demand for such numbers.

Curry has many more features which are useful to implement applications, like *set functions* [7] to encapsulate search, and standard features from functional programming, like modules or monadic I/O [37]. However, the kernel of Curry described so far should be sufficient to understand the remaining contents of this paper.

[4] The concrete syntax is simplified by omitting the declaration of free variables, like `x` and `xs`, which is required in Curry programs to enable consistency checks by the compiler.
[5] We use common notations from term rewriting [9].

Early implementations of functional logic languages, like PAKCS [3] or TOY [29], used Prolog as a target language due to its built-in support for non-determinism. A drawback of this approach is that they inherit the incompleteness of Prolog's backtracking strategy. In order to get rid of this fixed search strategy, subsequent implementations are based on the idea to represent non-deterministic choices as data. Instead of directly evaluating non-deterministic branches, the alternatives are returned as a tree structure so that search strategies can be defined as tree traversals, which supports an easy switch between different strategies. For instance, the Curry system KiCS2 [13] and Curry2Go [11] have options to select different search strategies, like depth-first, breadth-first, or fair search.

## 3 From Predicates to Functions

This section discusses methods to transform logic programs into functional logic programs by mapping predicates and goals into functions and nested expressions. Since predicates can be viewed as Boolean functions, the simplest transformation maps each predicate into a Boolean function and each clause into a (conditional) equation. For instance, the clauses of predicate `app` shown in Example 1 can be transformed into

```
app [] ys ys = True
app (x:xs) ys (x:zs) | app xs ys zs
                     = True
```

This *conservative transformation* [17] does not change the structure of derivations since narrowing steps on Boolean functions correspond to resolution steps. Thus, there is no real advantage to perform this transformation.

To exploit the computational power of functional logic languages, predicates should be transformed into non-Boolean functions by selecting some arguments as results and generating function definitions according to this selection.

**Example 4.** Consider again predicate `app` of Example 1. If the third argument is selected as a result argument (as often intended in logic programs), the clauses of `app` can be transformed into the following functional logic program:

```
app [] ys = ys
app (x:xs) ys | zs =:= app xs ys
              = x:zs
```

Although any set of argument positions can be selected as results, there are heuristics to select result arguments so that optimal evaluations are ensured for large classes of programs, as discussed in [17].

It is shown in [17] that, even if this *functional transformation* is used, there is a strong one-to-one correspondence, independent of the selection of result arguments, between resolution derivations w.r.t. the original logic program and narrowing derivations w.r.t. the transformed program. To improve this situation and get some computational advantage, one has to replace the unification occurring in conditions by *let* bindings whenever possible[6] and inline these bindings if reasonable. For instance, one can transform the rule

```
app (x:xs) ys | zs =:= app xs ys
              = x:zs
```

into

```
app (x:xs) ys = let zs = app xs ys
                in x:zs
```

and inline the binding of zs into

```
app (x:xs) ys = x : app xs ys
```

This *demand functional transformation* is described in detail in [17]. If the transformed program is eagerly evaluated, i.e., the arguments of a function call are evaluated before replacing the function call by its body ("call by value"), there is no operational difference between programs transformed by the functional and the demand functional transformation. This situation changes when the arguments are evaluated "by need," as in Haskell or Curry and discussed in [25, 26].

**Example 5.** Consider the predicate siglist defined by the clauses

```
siglist([],zero).
siglist([_],one).
siglist([_,_|_],many).
```

The demand functional transformation yields

```
siglist []      = Zero
siglist [_]     = One
siglist (_:_:_) = Many
```

Now consider the evaluation of the expression siglist (app $xs$ $ys$), where $xs$ is a long list with $n$ elements. An eager evaluation requires $n+2$ rewrite steps, whereas a non-strict language needs at most three steps.

[6] This is possible when the variable in the left-hand side of the unification has no occurrences in result arguments of other goal literals, see [17] for a precise discussion.

Although it seems that the demand functional transformation is the way to go, there is one potential problem of this transformation: it might change the semantics, i.e., the set of computed solutions. This could be the case if the evaluation of some subexpression is not demanded and its evaluation would fail to yield a value. This failure would be propagated in the original logic program, but it might be "hidden" in the transformed program. For instance, consider a predicate relating a non-empty list with its tail

```
tail([_|Xs],Xs).
```

and its application in the predicate

```
sigtail(S) :-
  tail([],Xs),
  app([0,1],Xs,Ys),
  siglist(Ys,S).
```

Due to the failure of the first subgoal, the goal "?- sigtail(S)." fails. However, the demand functional transformation yields

```
tail (_:xs) = xs
sigtail = siglist (app [0,1] (tail []))
```

The demand-driven or lazy evaluation of sigtail, which performs only necessary reductions, yields the value Many. This is conform to the mathematical principle of "replacing equals by equals" but it changes the set of answers w.r.t. the original logic program.

It is possible to modify the transformation so that the transformed functional logic program computes only more general answers than the original logic program, i.e., each answer of the functional logic program is a generalization of an answer computed by the logic program. Computing more general answers is also preferable in pure logic programming since it results in smaller search spaces.

As shown in [19], the demand functional transformation yields programs so that needed narrowing is sound and complete w.r.t. the logical consequences of the logic program, where soundness requires that all functions are totally defined. Hence, if there are also partially defined functions, one has to ensure that every occurrence of such a function in a computation will eventually be evaluated. This can be obtained by a slight modification of the demand functional transformation which is called *fail-sensitive functional transformation*. If a rule's right-hand side contains an application $(f\ \ e)$ and the evaluation of the expression $e$ *might fail* to compute a value, i.e., it contains a

partially defined function, then this application is replaced by

$(f\ \$!\ e)$

"$!" denotes function application with a strict evaluation of argument $e$. Thus, if the evaluation of $e$ fails, the evaluation of $(f\ \$!\ e)$ fails.

For instance, the fail-sensitive functional transformation maps the definition of predicate `sigtail` into

```
sigtail =
  siglist $! (app [0,1] $! tail [])
```

Then the evaluation of `sigtail` leads to a failure due to the enforced evaluation of `(tail [])`. Note that the operator "$!" must be inserted at all places where a potentially failing expression occurs and not only where the failing expressions occurs first.

The fail-sensitive functional transformation requires information whether operations are totally defined. Since this is undecidable in general, one can approximate this property by splitting it into two parts: termination and non-occurrence of failures due to incomplete patterns, as visible in the definition of `tail`.

Termination of rewrite systems or functional programs is well-studied so that various techniques are available to approximate this property, e.g., [16, 27]. To approximate absence of failures, one could simply mark a function as failing if it is defined with an incomplete set of patterns or call a failing function in its right-hand side. This results in a fixpoint computation of this property. This simple approximation can be improved by considering the context of using failing functions in right-hand sides. For instance, the following function uses the failing function `tail` but it is totally defined since `tail` is called with a non-empty list:

```
tailOrEmpty []     = []
tailOrEmpty (x:xs) = tail (x:xs)
```

The tool described in [20] approximate the failing property of functions by approximating call types for functions, which ensures a fail-free evaluation, and using call types to approximate the failure status of functions. For instance, the call type of `tail` are all non-empty lists so that the call of `tail` in the second rule of `tailOrEmpty` does not cause a failure. Hence, `tailOrEmpty` is totally defined. In practice, only a few operations of larger programs have non-trivial call types, i.e., might fail on specific arguments.[7]

Exploiting such tools, one can implement the fail-sensitive functional transformation in three steps. First, the logic program is transformed with the demand functional transformation as described in [17]. Then, the generated program is analyzed with the failure-inference tool described in [20] (automated termination checks are currently omitted since it is seldom that operations generated from Prolog are completely defined but non-terminating). Based on the failure information, the transformed program is modified by replacing function applications $(f\ e)$ by $(f\ \$!\ e)$ whenever the expression $e$ might fail.

This transformation produces functional logic programs which compute the same or more general answers compared to the original logic programs. In the worst case (if all functions are possible failing), the same number of evaluation steps are performed, but in many other cases, the transformation reduces the number of computation steps (due to the optimality of needed narrowing) so that infinite search spaces might be reduced to finite ones. Thus, the transformation has no disadvantage but in some cases one gets considerable improvements.

To evaluate this transformation, we have implemented a tool performing the fail-sensitive functional transformation as described above.[8] Table 1 contains the results of executing various Prolog programs with SWI-Prolog and SICStus-Prolog and the Curry programs obtained by applying the fail-sensitive functional transformation with the Curry system KiCS2 [13]. KiCS2 compiles Curry programs into Haskell and uses the Glasgow Haskell Compiler (GHC 9.4.5) to generate machine code.[9] The examples, which can be found in the appendix, are small programs since larger Prolog programs are seldom logic

[7] This requires also the consideration of intended types. For instance, `app` is totally defined on lists, which are the intended arguments, although `app` fails on the unintended argument `42`. The consideration of type information is discussed in [19].

[8] The tool, available at `https://cpm.curry-lang.org/pkgs/prolog2curry-1.2.0.html`, is implemented as a Curry package for easy installation. A script together with all required tools is available as a docker image at `https://hub.docker.com/r/currylang/prolog2curry`. A web interface to this transformation system is available at `https://cpm.curry-lang.org/webapps/pl2curry/`.

[9] The benchmarks were executed on a Linux machine running Ubuntu 22.04 with an Intel Core i7-1165G7 (2.80GHz) processor with eight cores. The time is the total run time of executing a binary generated with the Prolog/Curry systems.

| Language: | Prolog | Prolog | Curry |
|---|---|---|---|
| System: | SWI 9.0.4 | SICStus 4.9.0 | KiCS2 3.1.0 |
| rev_4096 | 0.23 | 0.22 | 0.10 |
| tak_27_16_8 | 6.97 | 3.23 | 0.74 |
| ackermann_3_9 | 2.13 | 8.72 | 0.07 |
| pali_[] | $\infty$ | $\infty$ | 0.01 |
| siglist_app_0 | $\infty$ | $\infty$ | 0.01 |
| numleaves_7 | $\infty$ | $\infty$ | 0.01 |
| permsort_10 | 1.43 | 0.28 | 0.03 |
| permsort_11 | 16.16 | 1.38 | 0.08 |
| permsort_12 | 206.34 | 15.23 | 0.28 |

**Table 1:** Execution times (in seconds) of Prolog and generated Curry programs

programs—they often use non-declarative features. Such non-declarative features are either not necessary in functional logic programs (e.g., cuts are replaced by exploiting functional dependencies) or can be reformulated in a declarative manner (e.g., declarative monadic I/O, state monads).

The first three benchmarks are purely deterministic computations. `rev_4096` is the naive list reversal applied to a list of 4096 elements. `tak_27_16_8` applies the highly recursive `tak` function [30] to the values `(27,16,8)` in Peano representation. The Ackermann function, defined on Peano numbers, is applied to the Peano representation of `(3,9)`. For these functions, the demand-driven evaluation strategy has no real advantage since the values of all subexpressions are required. The situation is different in the next three benchmarks where the original logic program has an infinite search space and the transformed functional logic program has a finite search space, similarly to Example 1. `pali_[]` denotes the evaluation of `pali([],M)` (see Examples 2 and 3), `siglist_app_0` denotes the evaluation of the goal "`app(Xs,[],Zs), siglist(Zs,zero)`", and `numleaves_7` denotes the generation of all binary trees with seven leaves. Since Table 1 shows the time to compute *all* answers to the given goals, the Prolog systems do not terminate due to the infinite search spaces. The final benchmarks, permutation sort applied to lists containing 10, 11, and 12 decreasing Peano numbers, demonstrates the advantage of demand-driven evaluation even if the search space is finite. As discussed at various places [6, 18], the functional logic version explores permutations in a demand-driven manner so that not all permutations are actually generated. Thus, our transformation maps a "generate-and-test" algorithm into a more efficient "test-of-generate-as-demanded" algorithm with a lower complexity, as apparent from the benchmarks.

## 4 From Backtracking to Complete Search Strategies

As already mentioned, Prolog is based on backtracking to deal with "don't know" non-deterministic resolution steps. This was a way to deal with limited hardware resources when Prolog was invented. Since this strategy is fixed for Prolog [15], it has the unfortunate consequence that many non-logical features, like input/output, arithmetic, search-space pruning (cut), depend on this strategy so that it is not easy to change it in real-world applications of Prolog. However, in order to close the gap between theory and practice of logic programming, support a higher-level understanding of programs, and improve the situation when teaching logic programming, complete search strategies are necessary.

Curry does not fix backtracking or depth-first search so that implementations can support other search strategies—there are no language features depending on backtracking. For instance, PAKCS [3, 22] compiles into Prolog so that backtracking search is used. KiCS2 [13] compiles Curry programs into Haskell programs and represent the search space as a tree structure on which search strategies are defined so that one can switch between depth-first (DFS) or breadth-first search (BFS), among others. Curry2Go [11] compiles Curry programs into Go[10] programs. Go is a statically typed language with garbage collection and direct support for CSP-like concurrency

[10] `https://golang.org/`

| Example | PAKCS | KiCS2 DFS | KiCS2 BFS | Curry2Go DFS | Curry2Go BFS | Curry2Go FS |
|---|---|---|---|---|---|---|
| nrev_4096 | 6.29 | 0.10 | 0.10 | 0.85 | 0.85 | 0.85 |
| takPeano_24_16_8 | 56.78 | 0.12 | 0.12 | 8.05 | 7.98 | 7.76 |
| primesHO_1000 | 29.46 | 0.04 | 0.04 | 3.51 | 3.58 | 3.55 |
| psort_13 | 18.92 | 0.35 | 2.32 | 7.11 | 7.25 | 9.51 |
| addNum_2 | 0.18 | 0.24 | 0.57 | 0.28 | 0.29 | 0.28 |
| addNum_5 | 0.20 | 2.01 | 4.36 | 0.67 | 0.67 | 0.35 |
| addNum_10 | 0.24 | 11.83 | 16.84 | 1.53 | 1.54 | 0.54 |
| select_50 | 0.09 | 0.19 | 0.27 | 0.02 | 0.02 | 0.02 |
| select_100 | 0.27 | 4.13 | 4.80 | 0.06 | 0.06 | 0.03 |
| select_150 | 0.56 | 25.10 | 32.42 | 0.13 | 0.13 | 0.06 |
| isort_primes4 | 9.56 | 0.02 | 0.02 | 1.15 | 1.14 | 1.11 |
| psort_primes4 | 112.38 | 0.02 | 0.02 | 1.11 | 1.11 | 0.71 |

**Table 2:** Comparing Curry system with search strategies

[24] and lightweight threads (*goroutines*). The latter feature is used to provide, in addition to DFS and BFS, a fair search strategy. For instance, consider the following contrived example:

```
idND :: a → a
idND n = loop ? n ? loop
```

where the evaluation of `loop` does not terminate. Semantically, `idND` is the identity function but, operationally, it is non-deterministically defined with looping alternatives. Both DFS and BFS loop on the expression `(idND 0)` instead of returning the value `0`, since there is no choice when evaluating `loop`. However, the *fair search* (FS) strategy of Curry2Go returns this value since FS evaluates non-deterministic branches concurrently as goroutines and collects the computed results in a channel [11].

To show that the efficiency of advanced search strategies is not really worse than backtracking, we compared these Curry implementations and their search strategies. Table 2 shows the run times (in seconds as the average of three runs) of various examples[11] and search strategies. The first three benchmarks are typical purely functional programs. `nrev_4096` is the quadratic naive reverse algorithm applied to a list with 4096 elements, `takPeano` is a highly recursive function on naturals [30] applied to arguments (24,16,8) in Peano representation, and `primesHO_1000` computes the 1000th prime number by constructing an infinite list of all primes via the sieve of Eratosthenes (using higher-order functions). For these examples, Curry2Go is much faster than PAKCS but less efficient than KiCS2, which is not surprising since Haskell/GHC is a highly optimized functional programming system.

The remaining non-deterministic benchmark programs show that KiCS2 and Curry2Go are competitive with PAKCS (which exploits Prolog's built-in support for non-determinism). `psort_13` is the naive permutation sort applied to a list of 13 elements. `addNum_`$n$ non-deterministically chooses a number (out of 2000) and adds it $n$ times, and `select_`$n$ non-deterministically selects an element in a list of length $n$ and sums up the element and the list without the selected element. The considerable slowdown in KiCS2 with increasing values for $n$ is caused by the duplication of choices in pull-tab steps [1] when non-deterministic expressions are shared, as discussed in [21]. Curry2Go avoids this problem by adding a kind of memoization for choices, as described in [11, 21].

Apart from the fact that the fair search strategy of Curry2Go is the only operationally complete strategy (e.g., it is able to compute a value of `idND 0`), there are also other interesting differences between the search strategies. For instance, KiCS2 shows some overhead of BFS compared to DFS (possibly due to the additional structures used to implement breadth-first tree search), whereas there is almost no overhead in Curry2Go (since the difference between BFS and DFS is just a different scheduling of tasks). Moreover, the fair search (FS) strategy is sometimes faster than BFS and DFS thanks to the use of goroutines possibly scheduled on different processors. This is also visible in the last two lines of Table 2 which show the time to sort the list

[11] The examples are available at `https://github.com/curry-language/curry2go`.

```
[ primes!!303, primes!!302
, primes!!301, primes!!300]
```

with the deterministic insertion sort (`isort`) and the non-deterministic permutation sort (`psort`) algorithm, respectively, where `primes` defines the infinite list of all prime numbers. Due to backtracking, identical computations might be repeated if they occur in different non-deterministic branches. Thus, `primes` is re-evaluated by PAKCS several times when the list is passed to the non-deterministic operation `psort`. This is not the case in implementations which represent choices in a graph structure so that the results of deterministic computations are shared across non-deterministic evaluations [12].

## 5 Conclusions

We have shown the advantage of using functions instead of predicates by presenting a systematic method to transform logic programs into functional logic programs so that the transformed functional logic programs always computes the same or more general answers than the original programs. This transformation does not introduce any operational disadvantage: in the worst case, the number of computation steps in the original and the transformed programs are identical, but in many other cases the number of computation steps is reduced and infinite search spaces are transformed into finite ones. Furthermore, we showed that applying complete search strategies on functional logic programs is competitive to backtracking search so that one get rid of the usual problems caused by backtracking. This closes the gap between theory and practice of logic programming and could lead to a higher, really declarative programming style. With these techniques, we can keep the ideas and advantages of logic programming in future programming systems beyond the restriction to predicates and backtracking.

## References


[1] S. Antoy. “On the Correctness of Pull-Tabbing”. In: *Theory and Practice of Logic Programming* 11.4-5 (2011), pp. 713–730.

[2] S. Antoy, R. Echahed, and M. Hanus. “A Needed Narrowing Strategy”. In: *Journal of the ACM* 47.4 (2000), pp. 776–822.

[3] S. Antoy and M. Hanus. “Compiling Multi-Paradigm Declarative Programs into Prolog”. In: *Proc. International Workshop on Frontiers of Combining Systems (FroCoS’2000)*. Springer LNCS 1794, 2000, pp. 171–185.

[4] S. Antoy and M. Hanus. “Declarative Programming with Function Patterns”. In: *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR’05)*. Springer LNCS 3901, 2005, pp. 6–22.

[5] S. Antoy and M. Hanus. “Default Rules for Curry”. In: *Theory and Practice of Logic Programming* 17.2 (2017), pp. 121–147.

[6] S. Antoy and M. Hanus. “Functional Logic Programming”. In: *Communications of the ACM* 53.4 (2010), pp. 74–85.

[7] S. Antoy and M. Hanus. “Set Functions for Functional Logic Programming”. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP’09)*. ACM Press, 2009, pp. 73–82.

[8] L. Augustsson et al. “The Verse Calculus: A Core Calculus for Deterministic Functional Logic Programming”. In: *Proc. ACM International Conference on Functional Programming (ICFP 2023)*. 2023, 203:1–203:31.

[9] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[10] J. Backus. “Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs”. In: *Comm. of the ACM* 21.8 (1978), pp. 613–641.

[11] J. Böhm, M. Hanus, and F. Teegen. “From Non-determinism to Goroutines: A Fair Implementation of Curry in Go”. In: *Proc. of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021)*. ACM Press, 2021, 16:1–16:15.

[12] B. Braßel and F. Huch. “On a Tighter Integration of Functional and Logic Programming”. In: *Proc. APLAS 2007*. Springer LNCS 4807, 2007, pp. 122–138.

[13] B. Braßel et al. “KiCS2: A New Compiler from Curry to Haskell”. In: *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*. Springer LNCS 6816, 2011, pp. 1–18.

[14] J. Cohen. “A View of the Origins and Development of Prolog”. In: *Communications of the ACM* 31.1 (1988), pp. 26–36.

[15] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog - the standard: reference manual*. Springer, 1996.

[16] J. Giesl et al. “Automatic Termination Proofs for Haskell by Term Rewriting”. In: *ACM Transactions on Programming Languages and Systems* 33.2 (2011), Article 7.

[17] M. Hanus. “From Logic to Functional Logic Programs”. In: *Theory and Practice of Logic Programming* 22.4 (2022), pp. 538–554.

[18] M. Hanus. “Functional Logic Programming: From Theory to Curry”. In: *Programming Logics - Essays in Memory of Harald Ganzinger*. Springer LNCS 7797, 2013, pp. 123–168.

[19] M. Hanus. “Improving Logic Programs by Adding Functions”. In: *Proceedings of the 34th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2024)*. Springer LNCS 14919, 2024, pp. 27–44.

[20] M. Hanus. “Inferring Non-Failure Conditions for Declarative Programs”. In: *Proc. of the 17th International Symposium on Functional and Logic Programming (FLOPS 2024)*. Springer LNCS 14659, 2024, pp. 167–187.

[21] M. Hanus and F. Teegen. “Memoized Pull-Tabbing for Functional Logic Programming”. In: *Proc. of the 28th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2020)*. Springer LNCS 12560, 2021, pp. 57–73.

[22] M. Hanus et al. *PAKCS: The Portland Aachen Kiel Curry System*. Available at `http://www.informatik.uni-kiel.de/~pakcs/`. 2023.

[23] M. Hanus (ed.) *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Available at `http://www.curry-lang.org`. 2016.

[24] C.A.R. Hoare. “Communicating sequential processes”. In: *Communications of the ACM* 21.8 (1978), pp. 666–677.

[25] G. Huet and J.-J. Lévy. “Computations in Orthogonal Rewriting Systems”. In: *Computational Logic: Essays in Honor of Alan Robinson*. Ed. by J.-L. Lassez and G. Plotkin. MIT Press, 1991, pp. 395–443.

[26] J. Hughes. “Why Functional Programming Matters”. In: *Research Topics in Functional Programming*. Ed. by D.A. Turner. Addison Wesley, 1990, pp. 17–42.

[27] C.S. Lee, N.D. Jones, and A.M. Ben-Amram. “The Size-Change Principle for Program Termination”. In: *ACM Symposium on Principles of Programming Languages (POPL’01)*. 2001, pp. 81–92.

[28] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[29] F. López-Fraguas and J. Sánchez-Hernández. “TOY: A Multiparadigm Declarative System”. In: *Proc. of RTA’99*. Springer LNCS 1631, 1999, pp. 244–247.

[30] W. Partain. “The nofib Benchmark Suite of Haskell Programs”. In: *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer, 1992, pp. 195–202.

[31] S. Peyton Jones, ed. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[32] U.S. Reddy. “Narrowing as the Operational Semantics of Functional Languages”. In: *Proc. IEEE Internat. Symposium on Logic Programming*. Boston, 1985, pp. 138–151.

[33] U.S. Reddy. “Transformation of Logic Programs into Functional Programs”. In: *Proc. IEEE Internat. Symposium on Logic Programming*. Atlantic City, 1984, pp. 187–196.

[34] J.A. Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *Journal of the ACM* 12.1 (1965), pp. 23–41.

[35] J.R. Slagle. “Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity”. In: *Journal of the ACM* 21.4 (1974), pp. 622–642.

[36] L. Sterling and E. Shapiro. *The Art of Prolog*. 2nd. Cambridge, Massachusetts: MIT Press, 1994.

[37] P. Wadler. “How to Declare an Imperative”. In: *ACM Computing Surveys* 29.3 (1997), pp. 240–263.

## Appendix

## A Source Code of Benchmarks

This appendix shows the Prolog source code of some predicates used in the benchmarks in Sect. 3 and the Curry code generated by our tool implementing the fail-sensitive functional transformation.

### A.1 `rev`

The predicate `rev` is the well-known naive reverse with a quadratic complexity:

```
app([],Xs,Xs).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

rev([],[]).
rev([X|Xs],R) :-
  rev(Xs,Zs),
  app(Zs,[X],R).
```

Both predicates are translated into totally defined functions:

```
app []     ys = ys
app (x:xs) ys = x : app xs ys

rev []     = []
rev (x:xs) = app (rev xs) [x]
```

### A.2 `tak`

The function `tak` is defined in Prolog on Peano numbers, where `o` represents zero and `s` represents the successor of a natural number:

```
tak(X,Y,Z,A) :-
  leq(X,Y,XLEQY),
  takc(XLEQY,X,Y,Z,A).

takc(true,X,Y,Z,Z).
takc(false,X,Y,Z,A) :-
  dec(X,X1),
  tak(X1,Y,Z,A1),
  dec(Y,Y1),
  tak(Y1,Z,X,A2),
  dec(Z,Z1),
  tak(Z1,X,Y,A3),
  tak(A1,A2,A3,A).

dec(s(X),X).

leq(o,_,true).
leq(s(_),o,false).
leq(s(X),s(Y),R) :- leq(X,Y,R).
```

Due to the definition of `dec`, the generated function `takc` contains occurrences of “`$!`” in its right-hand side:

```
tak x y z = takc (leq x y) x y z

takc True x y z = z
takc False x y z =
  ((tak $! (tak $! (dec x)) y z)
      $! (tak $! (dec y)) z x)
    $! (tak $! (dec z)) x y

dec (S x) = x

leq O     _     = True
leq (S _) O     = False
leq (S x) (S y) = leq x y
```

### A.3 `ackermann`

The Ackermann function is also defined as a Prolog predicate on Peano numbers, as presented in [36]:

```
ackermann(o,N,s(N)).
ackermann(s(M),o,Val) :-
  ackermann(M,s(o),Val).
ackermann(s(M),s(N),Val) :-
  ackermann(s(M),N,Val1),
  ackermann(M,Val1,Val).
```

It is translated into the Curry function

```
ackermann O     n     = S n
ackermann (S m) O     = ackermann m (S O)
ackermann (S m) (S n) =
  ackermann m (ackermann (S m) n)
```

### A.4 `numleaves`

The predicate `numleaves` relates a binary tree with the number of its leaves in Peano representation:

```
plus(o,Y,Y).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).

numleaves(leaf(_),s(o)).
numleaves(node(M1,M2),s(N)) :-
  numleaves(M1,N1),
  numleaves(M2,N2),
  plus(N1,N2,N).
```

This is translated into the Curry functions

```
plus O y = y
plus (S x) y = S (plus x y)

numleaves (Leaf _)     = S O
numleaves (Node m1 m2) =
  S (plus (numleaves m1) (numleaves m2))
```

### A.5 permsort

The permutation sort example computes permutations by non-deterministically inserting an element into a list.

```
% Non-deterministic list insertion:
insert(X,[],[X]).
insert(X,[Y|Ys],[X,Y|Ys]).
insert(X,[Y|Ys],[Y|Zs]) :-
  insert(X,Ys,Zs).

% Permutations:
perm([],[]).
perm([X|Xs],Zs) :-
  perm(Xs,Ys),
  insert(X,Ys,Zs).

% less-or-equal relation
leq(o,_).
leq(s(X),s(Y)) :- leq(X,Y).

% Is the argument list sorted?
sorted([]).
sorted([_]).
sorted([X,Y|Ys]) :-
  leq(X,Y),
  sorted([Y|Ys]).

% Permutation sort:
% search for some sorted permutation
psort(Xs,Ys) :- perm(Xs,Ys), sorted(Ys).
```

The generated operations `insert` and `perm` are totally defined, whereas `leq`, `sorted`, and `psort` might fail. Due to the strict left-to-right semantics of the predefined conjunction operator “`&&`”, insertions of the strict application operator “`$!`” in the third rule of `sorted` are not necessary.

```
insert x []       = [x]
insert x (y : ys) = x : (y : ys)
insert x (y : ys) = y : insert x ys

perm []       = []
perm (x : xs) = insert x (perm xs)

leq O _ = True
leq (S x) (S y) | leq x y
                = True
```

```
sorted []       = True
sorted [_]      = True
sorted (x:y:ys) | leq x y && sorted (y:ys)
                = True

psort xs | sorted ys = ys
  where ys = perm xs
```

# Multi-precision integer arithmetics

M. Anton Ertl
TU Wien
anton@mips.complang.tuwien.ac.at

## Abstract

Multi-precision integer arithmetics is widely used, among other things in public-key cryptography and when computing many digits of transcendental numbers. The present paper discusses multi-precision addition and multiplication: architectural support and its use in hand-written assembly language, libraries that use such assembly-language code, and programming language support and how close the code generated by Clang and GCC is to the hand-written assembly language.

## 1 Introduction

Integer arithmetics on numbers that fit in a single machine word is single-precision arithmetics. With two machine words, it's double-precision arithmetics.[1]. With more machine words, it's multi-precision arithmetics.

In the present work, I focus on multi-precision addition and multi-precision multiplication. Subtraction can be derived from addition, while division would merit a paper of its own. I also focus on unsigned arithmetic; for the common twos-complement representation of signed numbers signed addition and multiplication is the same (addition) or very similar (multiplication).

This paper is a first try at overview of uses and implementation techniques for efficient multi-precision integer arithmetics. The relevance of the topic can be seen from the use of multi-precision arithmetics in software (Section 2). Section 3 introduces the mathematical foundation, concepts, and notation for multi-precision arithmetics. Section 4 explains how various computer architectures support multi-precision arithmetics. This architectural support can be used to write routines in assembly language (Section 6) and provide them to application programs through libraries (Section 5), or programming language compilers can make use of these features and provide language support for them (Section 7).

[1] Single and double precision are more commonly used in connection with floating-point, but the concept also applies to integer arithmetics

## 2 Uses

How relevant is multi-precision arithmetic? In particular, how relevant is the performance of multi-precision arithmetic?

We can try to answer this question by looking at uses of the GMP (GNU multi-precision) library. E.g., 1839 packages (out of 65523, i.e., 2.8%) in Debian 12 depend on GMP. Unfortunately, that does not tell us anything about how relevant the performance of multiprecision arithmetics is for these packages. The GMP implementors invest a lot of effort into making GMP efficient; e.g., GMP 6.3.0 contains 155,715 lines in 930 files of assembly language code. This makes it plausible that multi-precision arithmetic in general and GMP in particular are used in applications where performance matters.

There are two applications dmains that I know where multi-precision arithmentics is used and where performance matters:

- Public-key cryptography uses arithmetic with sizes of 256–4096 bits in pre-quantum cryptography.[2]
- Computing many digits of transcendental constants such as $\gamma$ or $\pi$, which involve large multiplications. The numbers involved in these computations are bigger than the caches, so memory bandwidth becomes an issue; nevertheless, computing is also an issue.[3] A well-known program for this purpose is y-cruncher.

Another use is in implementing the integers of higher-level languages, which often support arbitrary size. In most programs this support is hardly used and is only a safety net in case the computation overflows a single-precision integer, but programs can also use this support as a convenient way of performing computations with big numbers.

## 3 Basics

We can use the same algorithms for synthesizing multi-precision arithmetics out of single and double-precision arithmetics as has been used for multi-digit arithmetics, so we use the same notation: A machine word can hold $b = 2^{\text{bits}}$ values: $0...b-1$. Single-precision numbers typically start with $s$, double-precision numbers with $d$, and multi-precision numbers with $m$. Double-precision numbers are represented with two machine words $d = s_1 b + s_0$, multi-precision numbers with $l$ machine words $m = s_{l-1} b^{l-1} + ... s_0$. Each $s_i$ is called a *limb* of $m$. In code fragments that deal with a single limb of a multiprecision number mx, we call that limb as `mx`, too.

### 3.1 Addition

When adding $n$ machine words, the result $r$ may be $\geq b$, but $r < nb$. So $r$ can be represented as $r = cb + s$, where $s$ is the part of $r$ that fits in a machine word and $c < n$ is the *carry*.

Multi-precision addition is performed starting from the least significant limbs and progresses towards more significant limbs. After the first step, the carry has to be added in addition to the limbs of the summands (*carry-in*): $r_i = \text{s1}_i + ... + \text{sn}_i + c_{i-1}$. For the carry-out of such a step, $c < n$ still holds.

### 3.2 Multiplication

When we multiply two single-precision numbers, the result has values of up to $b^2 - 2b + 1$ and fits in a double-precision number. This widening multiplication is often a primitive in implementing multi-precision arithmetics, and we will use it as such here.

For multi-precision multiplication $ef$, with $e$ having limbs $e_{l-1}...e_0$ and $f$ having limbs $f_{n-1}...f_0$, the result is

$$r = \sum_{i=0}^{l-1} \sum_{j=0}^{n-1} e_i f_j b^{i+j}$$

Here is an example showing the terms of a multiplication with $l = 4, n = 3$:

$$\begin{array}{cccccc} & & e_3 f_0 & e_2 f_0 & e_1 f_0 & e_0 f_0 \\ & e_3 f_1 & e_2 f_1 & e_1 f_1 & e_0 f_1 & \\ e_3 f_2 & e_2 f_2 & e_1 f_2 & e_0 f_2 & & \end{array}$$

All the terms in one column are for the same $i + j$. This table looks like the usual way long multiplication is presented, but note that each array entry is a double-precision result of a multiplication, unaffected by surrounding multiplications, whereas the usual long multiplication algorithm shows, in each line, the digits coming out of the more complex operation $ef_i$. So all the widening single-precision multiplications are independent of each other.

For computing a single limb $r_k$, we have to sum up a number of terms, and there is also a carry-out of this computation:

$$c_k b + r_k = \left( \sum_i (e_i f_{k-i}) \bmod b \right) + \left( \sum_i (e_i f_{k-i-1})/b \right) + c_{k-1}$$

The addition for computing a single limb can have up to $l + n + 1$ terms, with the carry being corresponingly large.

However, with different primitives, some of the additions can be integrated with the multiplications,

[2] Post-quantum cryptography often uses larger keys, but I do not know which of those approaches use multi-precision arithmetics. In any case, the keys still fit in the caches of modern processors.

[3] `https://www.numberworld.org/y-cruncher/faq.html#gpu` discusses some of the issues.

which can reduce the later summing. In particular, one can add two single-precision numbers to the double-precision result of a widening multiplication without overflowing the double-precision number: the maximum result of this operation is the maximum double number:

$$(b-1)*(b-1)+2(b-1) = b^2-2b+1+2b-2 = b^2-1$$

## 4 Architectural support

This section discusses the architectural support that existing architectures have for multi-precision arithmetics and that one could add to architectures to improve that support:

### 4.1 Addition

The most common support for multi-precision addition is in the form of carry flags. Many architectures have one carry flag C. On most such architectures the common addition instruction only sets carry-out, and there is an additional add-with-carry instruction that adds carry-in as well as producing carry-out, typically with a latency of one cycle.

E.g., AMD64 (without ADX) has one carry flag, and one can perform addition of two $n$-word multi-precision numbers `m = m1+m2`:

```
L: movq (m1,i,8),tmp
   adcq (m2,i,8),tmp
   movq tmp, (m,i,8)
   incq i
   decq n
   jnz  L
```

Note that this loop was carefully written with instructions that do not trample over the carry flag (many AMD64 instructions do).

ARM A64 also has one carry flag, but both the carry-in and flag-setting (including carry-out) is optional:

| writes flags: no | writes flags: yes | | |
|---|---|---|---|
| add | adds | no | carry-in |
| adc | adcs | yes | carry-in |

Compilers often have problems dealing with a single carry flag, and while you can ask the compiler to use add with carry-in and carry-out, even when the compiler uses the appropriate instruction, the code around it is not so great (see Section 7.2.2).

The Intel ADX extension allows using the O(verflow) flag as a second carry flag in an instruction that uses the O flag as both carry-in and carry-out. The motivation for adding ADX seems to have been particularly for multi-precision multiplication [2]. A simple example of its use is to add three $n$-word numbers `m = m1+m2+m3`:

```
L: movq (m1,i,8),tmp
   adcxq (m2,i,8),tmp
   adoxq (m3,i,8),tmp
   movq tmp, (m,i,8)
   incq i
   decq n
   jnz  L
```

Here each addition chain has its own carry bit: `tmp=m1+m2` uses C, `tmp=m3+tmp` uses O.

A number of architectures have no carry flag, among them RISC-V. They typically require a five-instruction sequence with a minimum latency of three cycles to replace one add with carry-in and carry-out instruction. The addition `m = m1+m2` looks as follows on RISC-V:

```
L: ld   summand1, 0(m1p)
   ld   summand2, 0(m2p)
   add  tmp1, carry, summand1
   sltu carry1, tmp1, summand1
   add  sum, tmp1, summand2
   sltu carry2, sum, tmp1
   add  carry, carry2, carry1
   sd   sum, 0(mp)
   addi n, n, -1
   addi mp, mp, 8
   addi m2p, m2p, 8
   addi m1p, m1p, 8
   bnez n, L
```

Finally, in yet-unpublished work [1] I propose adding a carry bit (and another bit for indicating signed overflow) to every general-purpose register. These bits should be easier to manage in a compiler than the traditional carry flag, and they allow having more than one addition chain active at once.

```
L: ld   summand1, 0(m1p)
   ld   summand2, 0(m2p)
   add    tmp1, summand1, summand2
   addc   sum, tmp1, sum
   sd   sum, 0(mp)
   addi   n, n, -1
   addi mp, mp, 8
   addi m2p, m2p, 8
   addi m1p, m1p, 8
   bnez n, L
```

The proposal follows the typical RISC-V style of having instructions with two source and one destination register, and therefore splits the computation `sum = summand1 + summand2 + carry(sum)` into two instructions: the `add` adds summand1 and summand2, and `addc` adds the carry from the previous iteration (in the carry bit of `sum`) to `tmp1`. Note that the `add` is not in the dependency cycle from the last iteration, that involves only the `addc`, which I expect to take one cycle of latency.

### 4.2 Multiplication

Most architectures support widening multiplication, either in one instruction with a double-word result or in one instruction for the lower word of the result, and one instruction for the upper word of the result.[4]

An instruction that computes $d = s_a s_b + s_c$ (integer multiply-add) would be useful, but I am not aware that any architecture has it. ARM A64 has `madd`, an instruction that computes the bottom word of $d$, but the corresponding instruction `umaddh` for computing the most significant word of $d$ is missing. However, we will show how such an instruction could be used (see Section 6.3).

ARM A64 also has `umaddl`, which performs $s = h_a h_b + s_c$, where $h$ indicates a 32-bit value. Unfortunately, this instruction is not useful for multi-precision arithmetics, because one needs 4 widening 32-bit multiplications plus additional work to replace one widening 64-bit multiplication.

I also explored the option of having an instruction `m2add` that computes $d = s_a s_b + s_x + s_y$ or `madc` that computes $d = s_a s_b + (c_z b + s_z)$ (so that an `add` followed by an `madc` would be equivalent to `m2add`), as well as upper/lower instruction-pair variants of these instructions. However, instructions are usually implemented in out-of-order execution engines to wait for all operands; so such instructions would result in carry-to-carry latencies that include the latency of the multiplication, which may be undesirable.

## 5 Libraries

The programming language support for high-performance multi-precision arithmetics has not been satisfactory, so programmers have developed libraries for this purpose. Wikipedia lists 36 libraries that deal with arbitrary-precision integers[5].

The most well-known among them is GMP, which has first been released in 1991. It contains a lot of functions for various purposes. In the present context, the low-level functions (`mpn_`...[6]) are the most interesting ones, and among them these functions.

- $m = m_1 + m_2$: `mpn_add_n(m,m1,m2,n)`
- $m = sm_1$: `mpn_mul_1(m,m1,n,s)`
- $m = m + sm_1$: `mpn_addmul_1(m,m1,n,s)`
- $m = m_1 m_2$: `mpn_mul(m,m1,n1,m2,n2)`

These functions are usually written in assembly language, with the central code similar to what is shown in Section 6, but with lots of macros and parameterization for unrolling and to generate code for similar functions, and are therefore not as easy to understand as one might like.

We take a closer look at how to implement the first and third of these functions. The third is a better stepping stone for implementing long multiplication (`mpn_mul`) than the second, and it also illustrates some implementation problems and how to deal with them.

A disadvantage of libraries is that you can combine their functions by calling one after the other, but the sequence cannot be optimized. It will always cost as much as the two individual calls. E.g., while it is possible to implement $m = m + sm_1$ by first calling `mpn_mul_1` and then `mpn_add_n`, GMP provides `mpn_addmul_1` as a separate function, in order to support more efficient implementations.

[4] Some architectures have additional instructions for multiplication where one or both operands are signed, but these are outside the scope of this paper.

[5] https://en.wikipedia.org/w/index.php?title=List_of_arbitrary-precision_arithmetic_software&oldid=1305070005#Libraries

[6] https://gmplib.org/manual/Low_002dlevel-Functions

## 6 Hand-written assembler code

In the following we take a look at hand-written (hopefully optimal) code for the central part of one limb of three computations: $m = m_a + m_b$, $m = m_a + m_b + m_c$, $m = sm_a + m_b$. This provides a reference for comparing with the compiler-generated code we see in Section 7.

The first and third computation directly correspond to `mpn_add_n()` and `mpn_addmul_1()`. The three-summand addition is instructive because it shows the limits of having a single carry flag. It can also be seen as a simplified variant of a problem we have in $m = sm_a + m_b$; while it looks like there are only two summands here, we actually have to add, in each iteration, three words: the upper word of $sm_a$ from the previous iteration, the lower word of $sm_a$ from the present iteration, and $m_b$.

*Central part* means the computation without loading the source limbs and storing the the result limb, and without loop overhead. We show assembly language code for RISC-V, ARM A64, and AMD64. On RISC-V and ARM A64, the destination register is leftmost, on AMD64 rightmost (AT&T syntax). In this section we use variable names instead of registers, to make the code easier to follow.

The latency numbers given are based on realistic assumptions about the timing of the instructions in a core with wide out-of-order execution; in particular, `add`, `adcs` etc. have one-cycle latencies, and `mov` has a zero-cycle[7] latency.

### 6.1 Two-summand addition

For $m = m_a + m_b$:

```
RISC-V:
add sum, summand1, summand2
sltu carry1, sum, summand1
add sum, sum, carry
sltu carry2, sum, carry
add carry, carry1, carry2

ARM A64:
adcs sum, summand1, summand2

AMD64:
#summand1 is in sum
adc summand1, sum
```

[7] In many recent cores with out-of-order execution, `mov` is usually performed by register renaming

On ARM A64 and AMD64, carry is in the C flag before this and the new value of carry is in the C flag afterwards. The RISC-V code has three cycles of carry-to-carry latency, the ARM A64 and AMD64 codes have 1 cycle of carry-to-carry latency on typical implementations of these architectures.

#### 6.1.1 Kogge-Stone addition

Alexander Yee has implemented Kogge-Stone addition (normally a hardware adder design technique) in AVX-512.[8] This is pretty far from anything discussed in this paper, so the rest of this paper ignores this possibility. However, Yee reports that his AVX512-F code “is about 2x faster than the classic approach of chaining adc instructions on Skylake X.”. So we should look at whether SIMD instruction sets can be more profitably used to achieve the best performance before investing much time in tuning the performance of carry-flag-based approaches.

#### 6.1.2 Breaking dependencies

Two-summand and three-summand addition with chains of carries can be latency-bound on wide cores, with the loop-carried dependencies through carry forming the critical path. Michael S presents[9] and evaluates[10] a technique for converting a data dependency into an extremely predictable control dependency, which eliminates the dependency in case of a correct prediction (i.e., almost always).

```
   xorq carry, carry
L: movq (ma,i,8),sum
   xorq carry2, carry2
   addq (mb,i,8),sum
   setc carry2
   addq carry, sum
   jc   C
M: movq carry2, carry
   movq sum, (m,i,8)
   incq i
   decq n
   jnz  L
   ... finish ...
   ret
C: incq carry2
   jmp  M
```

[8] `http://www.numberworld.org/y-cruncher/internals/addition.html`
[9] `news:20250805014356.0000073c@yahoo.com`
[10] `news:20250806204333.00006869@yahoo.com`

This code starts a new dependecy chain by clearing carry2 at the start, which is then set to the carry of $\text{ma}_i + \text{mb}_i$. The old carry is then added to sum, and there is a low probability that this addition has a carry, which would need to be added to carry2. The carry-out of this addition is not added to carry2 in a data-dependent way (e.g., with `adc`) to avoid producing a data dependency cycle. Instead, the carry-out is checked with a conditional branch, and on correct prediction (the usual case for this very predictable branch) iteration $i+2$ uses nothing from iteration $i$ except loop counter updates, which can be reduced by unrolling.

While this loop reduces the dependence chains through carry, one iteration contains a lot of instructions, and therefore this code is likely to take more than one cycle per output word despite the reduction in dependencies. One way to counteract this may be to unroll the loop by a factor of $n$, and do the first $n-1$ original iterations with `adc` and only the last using the conditional branch, in order to achieve a better balance between general resource consumption and dependencies.

### 6.2 Three-summand addition

For $m = m_a + m_b + m_c$, things become more interesting:

```
RISC-V:
add m, ma, mb
sltu carry1, m, ma
add m, m, mc
sltu carry2, m, mc
add m, m, carry
sltu carry3, m, carry
add carry, carry1, carry2
add carry, carry, carry3

RISC-V with carry in GPRs:
# carry = carry(m)+carry(mab)
add  tmp, ma, mb
addc mab, tmp, mab
add  tmp, mab, mc
addc m, tmp, m

ARM A64:
# carry = carry1+C
adcs m1, ma, carry1
adc carry1, xzr, xzr
adds m2, mb, mc
adc  carry1, carry1, xzr
adds m, m1, m2
```

```
AMD64:
# carry = carry1+C
adc ma, mb
setb carry2
add $-1, carry1
adc mc, mb
mov carry2, carry1
#sum in mb

AMD64 with ADX:
#ma in m
adcx mb, m
adox mc, m
```

The RISC-V, ARM A64, and AMD64 code has three cycles of carry-to-carry latency, the ADX code has one cycle. For the AMD64 code the `setb` may not break the dependency on the earlier contents of `carry1`, and it may be useful to insert a `movl $0, carry1` before the `setb`.

In the ARM A64 variant, carry1 can have values 0–2, but the complete carry is the sum of carry1 (in a general-purpose register) and the C flag; the sum can only have values 0–2. In the AMD64 variant, carry1 can only be 0–1. One could use the approach used for AMD64 for ARM A64 and vice versa. For generalizing to more summands, the ARM A64 variant may be better.

Another approach for implementing three-summand addition is to combine two two-summand additions, but doing this naively would increase the number of stores and loads, and the loop overhead. Instead, one can unroll the loop to do two two-summand additions while the intermediate result fits in registers, and, at the unrolling boundaries, use the carry-handling approach used by the AMD64 code above:

```
ARM A64:
#carry = carry1+c
#minus1=-1
adcs mab0, ma0, mb0
adcs mab1, ma1, mb1
adcs mab2, ma2, mb2
adcs mab3, ma3, mb3
adc  carry2, xzr, xzr
adds xzr, carry1, minus1
adcs sum0, mab0, mc0
adcs sum1, mab1, mc1
adcs sum2, mab2, mc2
adcs sum3, mab3, mc3
mov  carry1, carry2
```

For an unrolling factor of $n$, this approach has a latency of $n + 2$ (if the CPU can perform two `adcs` per cycle). If the loads are arranged to be as late as possible and the stores to be as early as possible, the number of registers needed is $n + 3$ ($n + 4$ when using load-pair and store-pair), plus another 4 for addresses and loop control. The same approach can also be used on AMD64, with similar code, latency, and register requirements.

### 6.3 Multiplication step

For $m = sm_a + m_b$:

```
RISC-V:
mulhu tmp1, s, ma
mul   sma, s, ma
add   tmp2, sma, mb
sltu  tmp3, tmp2, sma
add   tmp1, tmp1, tmp3
add   m, carry, tmp2
sltu  tmp4, m, carry
add   carry, tmp1, tmp4

RISC-V with carry in GPRs:
mulhu tmp1, s, ma
mul   sma, s, ma
add   tmp2, sma, mb
addc  tmp1, tmp1, tmp2
add   m, carry, tmp2
addc  carry, tmp1, m

ARM A64:
# carry = carry1+C
umulh tmp1, s, ma
mul   sma, s, ma
adcs  tmp2, sma, mb
adc   carry2, tmp1, xzr
adcs  m, tmp2, carry1
mov   carry1, carry2

ARM A64 with umaddh:
# carry = carry1+C
umaddh carry2, s, ma, mb
madd   smab, s, ma, mb
adcs   m, smab, carry1
mov    carry1, carry2

AMD64:
#rax = ma
mulq s
addq mb, rax
adcq $0, rdx
addq carry, rax
adcq $0, rdx
mov  rdx, carry

AMD64 with ADX:
#rdx = s
#carry = carry1+C+O
mulx ma, m, carry2
adcx mb, m
adox carry1, m
mov  carry2, carry1
```

While the carry can be represented in one general-purpose register, in some cases it reduced the instruction count and sometimes the latency to leave a part of the carry in the C (and, with ADX, the O) flag, so I used this in some cases.

Concerning latency, on a wide-enough processor with out-of-order execution the execution speed is determined by loop-carried dependencies. In the loop surrounding these code fragments the `carry` register (or `carry1` and the C flag) are loop-carried, so the latency from the carry coming from the last iteration to the carry passed to the next iteration is relevant. Therefore this code has been written to first perform the multiplication and the addition of $m_b$, and only then perform the addition of `carry`. In the code above this latency has been minimized, in some cases at the cost of additional instructions; in some cases `mov` there is a mov instruction that can be eliminated by adding the carry-in earlier if latency is not an issue.

The carry-to-carry latency is: 3 cycles for RISC-V, 2 cycles for ARM A64, AMD64 and RISC-V with carry in GPRs, 1 cycle for AMD64 with ADX and ARM A64 with `umaddh`.

In the code with upper/lower pairs, I keep the pairs together in the same order produced by compilers, in the hope that the hardware combines the halves and reduces the resource consumption of the code (for RISC-V this is recommended). This results in an additional `mov` on the ARM A64 with `umaddh`.

## 7 Programming language support

### 7.1 High-level languages

Many high-level languages provide arbitrary-precision integers (aka bignums). However, most programs deal with integers small enough to fit

in one machine word all the time, and the arbitrary decision is just a better way (than terminating the program or modulo arithmetics) to deal with cases where the result of an operation does not fit in one machine word. Moreover, in these languages multi-word integers are dynamically sized and therefore implemented boxed, with dynamic memory allocation and usually with dynamic typing; all of these implementation techniques cost performance, so one would not implement a high-performance application like y-cruncher in such a language, or at least rewrite it in a low-level language such as “C/C++ with Intel SSE intrinsics”[11] soon.

It would be possible to improve the performance of multi-precision arithmetics in these languages, by a lot, but I don't expect it to happen anytime soon, thanks to the vicious circle of it not happening due to lack of demand and the existing demand switching to lower-level languages.

### 7.2 Low-level languages

In particular, I look at C, but the techniques should work in other low-level languages, too. The generated code depends on the compiler, however.

The code presented below is the output of clang 14.0.6 `-march=x86-64-v4` on AMD64, and clang 11.0.1 on ARM A64 and RV64GC (RISC-V), with -Os to reduce the amount of unrolling in order to get a result that is easier to understand and present; however, the compilers tend to be better in making use of the flag during one iteration than between iterations, so unrolling generally helps in more ways than just reducing the loop overhead.

I have also looked at the output of gcc in some cases, and have not seen significantly better results, except through unrolling.

#### 7.2.1 Double precision

64-bit integers have been supported on C implementations with 32-bit word size since around 1990. It took until around 2005 until 128-bit integers appeared in some gcc targets with 64-bit words (and then it took several years until all the operations on these integers worked correctly in gcc).

[11] `https://www.numberworld.org/nagisa_runs/euler-14.9b_log2-15.5b.html`

With these double-precision types available (we use `d_t` for double-precision and `s_t` for single-precision unsigned integers), we can use them to implement multi-precision. For two-summand multi-precision addition, the central loop is:

```
for (i=0; i<n; i++) {
    s_t s1=m1[i];
    s_t s2=m2[i];
    d_t d=s1+(d_t)s2+carry;
    m[i] = d;
    carry = d>>64;
}
```

The `d>>64` extracts the most significant word of `d`, and should be changed appropriately if the word size is not 64 bits.

On RISC-V the code above results in the canonical five-instruction sequence for add with carry-in and carry-out (Section 6.1) for the computation part of this loop. Unfortunately, the results on AMD64 and ARM A64 are far from optimal; for the central addition in this loop:

```
AMD64
r9=carry
xorl %eax, %eax
addq (%rsi,%r8,8), %r9
setb %al
addq (%rdx,%r8,8), %r9
adcq $0, %rax
#after storing r9:
movq %rax, %r9

ARM A64
x8=carry x9=s1 x10=s2
adds x8, x9, x8
adcs x9, xzr, xzr
adds x10, x8, x10
adcs x8, x9, xzr
```

The compilers obviously do not make optimal use of a carry flag for this code.

For three-summand addition, the C code with double precision is:

```
for (i=0; i<n; i++) {
    s_t s1=l1[i];
    s_t s2=l2[i];
    s_t s3=l3[i];
    d_t d=s1+(d_t)s2+(d_t)s3+carry;
    l[i] = d;
    carry = d>>64;
}
```

Note that carry can have the value 2 in this code, so in order for the compiler to use two carry flags (with ADX), it would have to represent that sum distributed over the two carry flags, and fuse the addition of each carry flag into a different one of the two addition of the three limbs. Not beyond today's compiler technology, but it would require an amount of development that apparently has not been invested yet, and probably never will. The code for two architectures is:

```
RV64GC             AMD64
#a6=carry          #r10=carry
#t1,a7,t0=s1/2/3
add  a5, t1, a6    xorl %eax, %eax
sltu a6, a5, t1    addq (%rsi,%r9,8), %r10
add  a7, a7, a5    setb %al
sltu a5, a7, a5    addq (%rdx,%r9,8), %r10
add  a6, a6, a5    adcq $0, %rax
add  t0, t0, a7    addq (%rcx,%r9,8), %r10
sltu a5, t0, a7    adcq $0, %rax
add  a6, a6, a5    #after storing r10:
                   movq %rax, %r10
```

The RISC-V code has the same number of instructions as the hand-written code, but does not minimize the carry-to-carry latency; It may be possible to fix that aspect by source-code changes.

For ARM A64 the code is slightly worse than the hand-written code, for AMD64 significantly worse. Concerning ADX, clang does not generate ADX instructions with the options I used, possibly because ADX has not been included in x86-64-v4. The ARM A64 code follows the pattern of the two-summand addition, i.e., it now has 6 instructions.

The central loop of $m = sm_a + m_b$ can be written as:

```
for (i=0; i<n; i++) {
    s_t s1=ma[i];
    s_t s2=mb[i];
    d_t d = (s*(d_t)s1+s2)+carry;
    m[i] = d;
    carry = d>>64;
}
```

Here carry is a full word. Here's what clang produces for the central computation:

```
RV64GC
#a6=carry
#t0=s1 t1=s2
mulhu t2, t0, a1
mul   t0, t0, a1
add   a5, t1, a6
sltu  a6, a5, t1
add   t0, t0, a5
sltu  t1, t0, a5
add   a5, a6, t2
add   a6, a5, t1

ARM A64:
#x8=carry
#x11=s1 x12=s2
umulh x13, x11, x1
adds  x8, x12, x8
mul   x11, x11, x1
adcs  x12, xzr, xzr
adds  x11, x8, x11
adcs  x8, x12, x13
#x11=bottom(d)

AMD64
#rax=carry
mulxq (%r9,%r10,8), %rbx, %r11
xorl  %esi, %esi
addq  (%rcx,%r10,8), %rax
setb  %sil
addq  %rbx, %rax
adcq  %r11, %rsi
#after storing rax:
movq  %rsi, %rax
```

Again, the RISC-V code uses as many instructions as the hand-written version, but with worse carry-to-carry latency, which may be fixable by writing the source code a little differently. Likewise, the ARM A64 code uses as many instructions but worse carry-to-carry latency.

The AMD64 code also has a worse carry-to-carry latency than the hand-written code. It uses mulxq from the BMI2 extension (which is in x86-64-v4) to achieve more flexibility in register allocation. Compiling the C code to for straight AMD64 code results in an additional register-to-register move.

### 7.2.2 C with Carry

There are two ways to make carry explicit in some C compilers:

- The Intel-specific way is to use the intrinsic `c_out=_addcarry_u64(c_in, s1, s2, &sum)`
- The Clang-specific way is to use the builtin `sum= __builtin_addcll(s1, s2, c_in, &c_out)`

GCC does not have a builtin that supports carry-in.

I show the in the following, because it works for architectures other than IA-32 and AMD64.

For two-summand addition, the loop with the Clang-specific way looks as follows:

```
for (i=0; i<n; i++) {
  s_t s1=l1[i];
  s_t s2=l2[i];
  l[i] =
    __builtin_addcll(s1,s2,carry,&carry);
}
```

For RISC-V a five-instruction sequence for add with carry-in and carry-out is produced, slightly different from the one shown earlier.

For the other architectures, the sequences are:

```
ARM A64:
adds x9, x9, x10
cset w10, hs
adds x9, x9, x8
cset w8, hs
orr  w8, w10, w8

AMD64:
addq   (%rdx,%r8,8), %r9
setb   %r10b
addq   %rax, %r9
setb   %al
orb    %r10b, %al
movzbl %al, %eax
```

All these sequences fail to make use of the `adcs` (ARM A64) or `adc` (AMD64) instruction and produce longish, RISC-V-like workarounds.

The source code for the Intel-specific way differs only in the call to the intrinsic in the obvious way, and we only get code for AMD64:

```
addb $-1, %r8b
adcq (%rdx,%rax,8), %r9
setb %r8b
```

This code actually makes use of the `adc` instruction. However, because the compiler is pretty bad at preserving the C flag, it transfers it to a general-purpose register with `setb` right after the `adc` instruction, and transfers it back into the C flag with the `addb` right before it.

For the three-summand addition, using the Clang-specific builtin leads to two instances of the code shown above (with variations in register allocation) on all three architectures, which is worse than the code that uses double precision. This is particularly unexpected for those architectures that actually have a carry flag and an add with carry-in and carry-out.

When using the Intel-specific intrinsic, the result is also two instances of the three-instruction sequence shown above for this intrinsic, but that's not that much worse than the hand-written code.

### 7.2.3 `_BitInt`

The C23 standard allows defining integer types for which you can specify the number of bits, e.g, as follows:

```
typedef unsigned _BitInt(4096) u4096;
```

This feature is supported in gcc-15.1 and clang-20.1 on AMD64 (not on all architectures), with an implementation-dependent maximum number of bits. gcc-15.1 supports up to 65,535 bits, while clang-20.1 supports up to 8,388,608 bits. I looked at the code generated by these two compilers with `-Os` for a three-summand addition with 65535 bits.[12]

Clang produces straight-line code, which tends to make good use of the carry flag (no ADX instructions). However, the number of live words in this computation does not fit in the registers, and the register allocator deals with this situation not as well as one might like, resulting in about 9 instructions for each word of the result.

GCC produces a loop unrolled by a factor of 2, but unfortunately first performs the two additions from the first original iteration and then the ones from the second original iteration, so it has to move the carries between general-purpose registers and the carry flags all the time, in the way

[12] https://godbolt.org/z/a5c9c8non

that we see with the Intel intrinsic. Overall gcc executes 10 instructions per result word.

Still, these are early days for this language feature. Hopefully the generated code will improve over time. Certainly the potential exists: With this feature the programmer directly expresses the intended meaning, instead of expressing it through lower-level features such as double-precision integers, builtins or intrinsics, where the compiler may have to extract the intended meaning from this lowered code in order to transform it to the best code for the architecture (maybe Kogge-Stone addition using AVX-512).

## 8 Further Work

While the present work contains some performance estimations, actual measurements are missing. This is especially relevant given that the main focus in this paper has been latency, while resource constraints may limit performance more on today's processors.

## 9 Conclusion

The present work looks at several multi-precision integer operations, how AMD64, ARM A64 and RISC-V support implementing them and what the corresponding assembly language code looks like. These operations can implemented through libraries written in assembly language, or through programming languages. In particular, the present work looks at the code resulting from C with double-precision integers (`uint128_t`), C with a builtin or intrinsic for add with carry-in and carry-out, and C with `_BitInt(...)`. The code using double precision often works better than the code with intrinsics or (worst) builtins; while `_BitInt(...)` is not yet as good as one would naively expect, it has the best future potential.

## References


[1] M. Anton Ertl. "Extending General-Purpose Registers with Carry and Overflow Bits". Paper rejected from ARITH 2024. 2024. URL: http://www.complang.tuwien.ac.at/anton/tmp/carry.pdf.

[2] Erdinc Ozturk et al. *New Instructions Supporting Large Integer Arithmetic on Intel Architecture Processors*. White Paper 327831-001. Intel, 2012. URL: https://www.intel.cn/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf.

# Noo: Towards a Meta-Language Calculus

Björn Lötters
Technische Hochschule Mittelhessen
Fachbereich Mathematik, Naturwissenschaften und Informatik
Wiesenstraße 14, D–35390 Gießen
bjoern.loetters@mni.thm.de

## Abstract

Dependent types tend to become verbose rather quickly. Consequently, many languages that support dependent types also provide powerful syntax extension mechanisms. These mechanisms enable the definition of embedded domain-specific languages (DSLs), facilitating safe abstraction (cf. [5]). However, such mechanisms have their limitations. Declarative approaches, such as Agda's mixfix operators [1, 4], support only a subset of context-free languages. Procedural approaches, such as Lean's macro system [10], allow the definition of arbitrary context-free syntax but sacrifice much of the purity and elegance of context-free grammars.

Noo [8] is a dependently typed general-purpose programming language featuring dependent (co-) pattern matching in the style of Agda [3] and a novel syntax-extension formalism based on our prior research [7, 9]. At its core, this formalism relies on context-free grammars, enabling the unrestricted definition of context-free syntax within the language while preserving much of the purity of Chomsky's formalism [2, 6]. The language is structured into a meta-language and an object language, with the object language being unfolded through a bootstrapping process. This approach allows for the definition of arbitrary new object languages and, consequently, embedded domain-specific languages for more natural and safer abstractions.

## References

[1] Annika Aasa. "Precedences in specifications and implementations of programming languages". In: *Theoretical Computer Science* 142.1 (1995). Selected Papers of the Symposium on Programming Language Implementation and Logic Programming, pp. 3–26. ISSN: 0304-3975. URL: https://www.sciencedirect.com/science/article/pii/030439759590680J.

[2] Noam Chomsky. "On certain formal properties of grammars". In: *Information and Control* 2.2 (1959), pp. 137–167. ISSN: 0019-9958.

[3] Jesper Cockx and Andreas Abel. "Elaborating dependent (co)pattern matching". In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). URL: https://doi.org/10.1145/3236770.

[4] Nils Anders Danielsson and Ulf Norell. "Parsing Mixfix Operators". In: *Implementation and Application of Functional Languages*. Ed. by Sven-Bodo Scholz and Olaf Chitil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 80–99. ISBN: 978-3-642-24452-0.

[5] Paul Hudak. "Building domain-specific embedded languages". In: *ACM Comput. Surv.* 28.4es (Dec. 1996), 196–es. ISSN: 0360-

0300. URL: https://doi.org/10.1145/242224.242477.

[6] Lennart C.L. Kats, Eelco Visser, and Guido Wachsmuth. "Pure and declarative syntax definition: paradise lost and regained". In: *SIGPLAN Not.* 45.10 (Oct. 2010), pp. 918–932. ISSN: 0362-1340. URL: https://doi.org/10.1145/1932682.1869535.

[7] Björn Lötters. "Context-Free Subphrase Grammars". In: *Trends in Functional Programming*. Ed. by Jason Hemann and Stephen Chang. Cham: Springer Nature Switzerland, 2025, pp. 112–133. ISBN: 978-3-031-74558-4.

[8] Björn Lötters. *Noo Programming Language*. https://git.thm.de/theorem-prover/noo-programming-language. Accessed: 2025-07-15. 2025.

[9] Björn Lötters and Uwe Meyer. "Context-Free Binding Grammars". In: *Proceedings of the 36th Symposium on Implementation and Application of Functional Languages*. IFL '24. Association for Computing Machinery, 2025, pp. 25–37. ISBN: 9798400710254. URL: https://doi.org/10.1145/3723325.3723338.

[10] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 625–635. ISBN: 978-3-030-79876-5.

# Ein neuer Pattern Matching Ansatz in Java-TX

Daniel Holle
Baden-Wuerttemberg Cooperative State University
Horb, Germany
d.holle@hb.dhbw-stuttgart.de

## Abstract

Java-TX ist eine auf Java basierende Programmiersprache, die Java um verschiedene Konzepte aus funktionalen Programmiersprachen erweitert. Dazu gehören eine globale Typinferenz und echte Funktionstypen. Ein weiteres Feature, welches im Rahmen dieser Arbeit implementiert werden soll, ist das Pattern Matching. Hierzu soll zum einen der Stand von Java erreicht werden, das heißt ein Pattern Matching innerhalb von Switch-Statements soll ermöglicht werden. Neu hinzu kommt, dass Pattern Matching auch innerhalb von Funktionsköpfen angewandt werden darf. Dieses Feature ist von Haskell inspiriert. Durch die Anwendung auf eine objektorientierte Sprache mit Funktionsüberladung ergeben sich einige interessante Herausforderungen.

## 1 Motivation

Pattern Matching ist an sich eine erweiterte Fallunterscheidung. Im Gegensatz zu normalen If-Statements wird hier keine Bedingung überprüft, sondern nach der Struktur von Werten unterschieden. Diese Struktur wird über sogenannte Patterns definiert.

In der Mathematik werden Fallunterscheidungen oft eingesetzt, um Funktionen zu beschreiben, die auf bestimmten Untermengen des Definitionsbereichs anders definiert sind. Deshalb bildet Pattern Matching eine intuitive Implementierungsgrundlage für solche Funktionen und wird in der funktionalen Programmierung oft verwendet.

Ein Pattern kann zum Beispiel eine simple Equivalenz sein, die einen Wert überprüft, wie die Zahl 10 oder eine Zeichenkette „text". Andererseits können auch komplexere Patterns abgefragt werden, wie „eine Zeichenkette, die mit dem Buchstaben A anfängt", oder „eine Liste mit 3 Elementen". Außerdem kann ein Pattern ein Wertkonstruktor sein. Hierbei kann der konkrete Typ eines Wertes abgefragt werden und der Wert gegebenenfalls dekonstruiert werden.

In Haskell wird das über Datendefinitionen möglich gemacht, welche eine Menge von validen Werten definieren. Dies entspricht als algebraischer Datentyp einem Summentyp.

In Java wird ein alternativer Ansatz mithilfe von Vererbung verfolgt, um eine solche Fallunterscheidung möglich zu machen. Dieser wird in Abschnitt 2 näher erläutert.

```
f :: String -> Int
f "text"      = 1 -- Exakt "text"
f ('A' : rs) = 2 -- Beginnt mit A
f [a,b,c]     = 3 -- Drei Buchstaben
f _           = 4 -- Alles andere

data Option a =
  Some a | None

g :: Option Int -> Int
-- Some wird dekonstruiert
g (Some i) = i
g None = 0
```

**Listing 1:** Pattern Matching in Haskell

Da Haskell keine Vererbung besitzt, kann nicht ohne weiteres eine Funktion definiert werden, die sowohl `String` als auch `Option` akzeptiert. In

Java ist das über den gemeinsamen Supertyp `Object` möglich.

Wie aus dem Listing ersichtlich, besitzt Haskell eine Reihe von Sprachkonstrukten, um das Pattern Matching leichter und mächtiger zu machen. Diese sind in Java bislang noch nicht angekommen.

## 2 Pattern Matching in Java

In Java gab es von Anfang an Switch-Statements. Diese machen es möglich, nach Konstanten zu matchen. Da das Switch-Statement ursprünglich (in C) dazu vorgesehen war, einen möglichst effizienten Tableswitch als Sprachfeature zu verankern, besitzt das Switch-Statement ein paar Eigenheiten, die es schwierig machen, dieses Sprachkonstrukt für Pattern Matching zu erweitern.

Zunächst waren Switch-Statements nur für primitive ganzzahlige Datentypen vorgesehen. Dadurch war eine Übersetzung in einen Tableswitch noch sehr einfach und direkt. Mit Java 7 konnten nun auch Strings als Labels verwendet werden. Hier wurde ganz klar eine Richtung eingeschlagen, um das Switch-Statement in ein höheres Sprachfeature zu entwickeln, das nicht mehr direkt mit einem Sprungbefehl implementiert werden kann.

Ein weiteres Problem stellt der sogenannte „Fall-Through” dar. Vereinfacht gesagt funktioniert ein Case-Label wie eine Sprungmarke. Das heißt, alle Befehle unterhalb der Sprungmarke werden so lange ausgeführt, bis ein Break-Statement erreicht wird. Wenn nun aber die einzelnen Fälle sich gegenseitig ausschließen, muss garantiert werden, dass ein Break-Statement vor dem nächsten Case-Label folgt. Ansonsten ist das Programm fehlerhaft.

### 2.1 Records

Records sind ein Sprachfeature, welches in Java 16 eingeführt wurde [2]. Auf den ersten Blick sind Records lediglich Klassen, die von `java.lang.Record` erben und einige Methoden wie `toString` und `equals` automatisch implementieren. Eine wichtige Eigenschaft ist, dass Records nur finale Felder besitzen können. Ein Record ist damit immutable. Records eignen sich gut um Value-Klassen zu implementieren, also Klassen, die als simple Halter von Daten agieren und keinen eigenen Zustand besitzen.

Wichtig für das Pattern Matching ist, dass Records anders als normale Klassen einen kanonischen Konstruktor besitzen. Dieser wird in der Record-Definition angegeben und definiert sowohl den Konstruktor selber als auch die finalen Felder des Records. Der kanonische Konstruktor definiert, wie ein Record in einem Pattern dekonstruiert werden kann. Er entspricht damit funktional einem Wertkonstruktor in Haskell, der für das Pattern Matching verwendet wird. In [4] unter „Future Work” wird darauf verwiesen, dass Pattern Matching eventuell in Zukunft auch für normale Klassen verwendet werden kann.

```
record Point(Number x, Number y) {}

int m(Object o) {
  return switch(o) {
    case Point(int a, int b) ->
      a + b;
    case Point(double a, double b) ->
      (int) (a * b);
    case Point(var a, var b) ->
      a.intValue() - b.intValue();
    case String s -> s.length();
    default -> -1;
  };
}
```

**Listing 2:** Pattern Matching mit Records

Wie in Listing 2 zu sehen, wird hier jeweils eine Instanz vom Typ `Point` dekonstruiert. Die Felder, welche aus dem Record herausgenommen werden, entsprechen genau der Position innerhalb des Konstruktors von `Point`. Es kann ein beliebiger Variablenname verwendet werden. Der Typ muss nicht exakt übereinstimmen. Er kann, wie in diesem Beispiel, auch spezieller sein als in der Record-Definition angegeben. In diesem Fall wird eine weitere Unterscheidung zur Laufzeit gemacht, welche die Typen überprüft. Das zeigt, dass Patterns auch ineinander geschachtelt werden können. Wenn exakt derselbe Typ wie in der Record-Definition gewollt wird, kann auch das Schlüsselwort `var` verwendet werden.

Normale Klassen wie `String` können nicht dekonstruiert werden, hier wird lediglich der Typ gematcht.

### 2.2 Sealed Interfaces

Sealed Interfaces sind ein weiterer Baustein für das Pattern Matching. Diese werden in [3]

beschrieben. Wie auch Records können Sealed Interfaces in verschiedenen Kontexten sinnvoll sein. Für sich gesprochen, erlaubt ein Sealed Interface eine Reihe von Klassen anzugeben, welche das Interface implementieren dürfen. Damit ist dieser Interface-Typ abgeschlossen und kann nicht mehr von außerhalb erweitert werden. Das kann zum Beispiel für ein besseres API-Design verwendet werden.

Für das Pattern Matching ist interessant, dass innerhalb von Switch-Expressions nun garantiert werden kann, dass alle möglichen Fälle implementiert worden sind. Damit kann der Default-Case wegfallen. Falls eine neue Implementierung erlaubt wird, ist das Programm nun nicht mehr valide, bis alle Switch-Expressions angepasst worden sind. Das hilft Fehler zu vermeiden.

Pattern Matching mit Sealed Interfaces kann effektiv dafür verwendet werden, um das Visitor-Pattern [5] in Java zu ersetzen. Die Garantien durch den Compiler sind dieselben und im Gegensatz zum Visitor-Pattern kann die Implementierung an einer Stelle zusammengefasst werden, statt sie über mehrere Klassen zu verteilen.

```
sealed interface Shape
  permits Circle, Square {}

record Circle(int r)
  implements Shape {}
record Square(int w)
  implements Shape {}
```

**Listing 3:** Sealed Interfaces

## 2.3 Switch-Expression

Wegen der in Abschnitt 2 oben genannten Probleme mit Switch-Statements wurden die Switch-Expressions als neues Sprachkonstrukt eingeführt [1]. Diese sind, wie der Name schon vermuten lässt, echte Expressions. Das heißt, sie werden zu einem Wert ausgewertet. Es können mehrere Case-Labels zusammengefasst werden und der „Fall-Through" entfällt.

```
int input = ...;
int res;
switch(input) {
  case 10: case 20: case 30:
    res = 1;
    break;
  case 40: res = 2; break;
  default: res = 0;
}
```

```
res = switch(input) {
  case 10 | 20 | 30 -> 1;
  case 40 -> 2;
  default -> 0;
};
```

**Listing 4:** Switch-Statement & Switch-Expression

Wie hier zu sehen ist, ist die Switch-Expression kompakter und lässt sich einfacher lesen. Die Switch-Expression kann auch als Statement verwendet werden und muss keinen Wert zurückgeben. Sowohl das Switch-Statement als auch die Switch-Expression können grundsätzlich für das Pattern Matching eingesetzt werden. Weil die Switch-Expression jedoch kompakter ist, wird diese in den nächsten Beispielen verwendet.

## 2.4 Zusammenspiel der Features

Wenn Records, Sealed Interfaces und Switch-Expressions zusammen verwendet werden, kann im Folgenden die Funktion `g` aus Listing 1 implementiert werden:

```
sealed interface Option<T>
  permits Some, None {}

record Some<T>(T a)
  implements Option<T> {}
record None<T>()
  implements Option<T> {}

int g(Option<Integer> o) {
  return switch(o) {
    case Some(Integer a) -> a;
    case None() -> 0;
  };
}
```

**Listing 5:** Funktion g mit Pattern Matching

Das Interface `Option` agiert hier als übergeordneter Typ, während `Some` und `None` als Wertkonstruktoren fungieren. Durch das Sealed Interface ist der Typ `Option` abgeschlossen und der Default-Case kann wegfallen. Wie in Haskell kann `Option` nicht direkt instanziiert werden, wohl aber Objekte vom Typ `Some` und `None`. Anders als in Haskell können beide allerdings auch als Typen verwendet werden, was für Java-TX interessant ist, weil dadurch Funktionsüberladungen für die einzelnen Konstruktoren möglich sind.

## 3 Anpassungen an Java-TX

Für das Pattern Matching in Java-TX im Rahmen von Switch-Expressions wurde die Semantik aus Java mit wenig Änderungen übernommen. Um mit Funktionsdefinitionen kohärent zu bleiben, ist das `var`-Schlüsselwort bei verschachtelten Patterns optional.

In Java-TX können Methoden automatisch überladen werden, falls die Typinferenz mehrere Lösungen für dieselbe Funktion berechnet. Dazu passend soll in Java-TX das automatische Überladen von cases innerhalb von Switch-Expressions möglich sein.

```
import java.lang.Integer;
import java.lang.Double;
import java.lang.Number;

record R(Number n) {}

Number f(Double d) { ... }
Number f(Integer i) { ... }

public m(r) {
  return switch(r) {
    case R(o) -> f(o);
  };
}
```

**Listing 6:** Überladung von cases in Switch-Expressions

Durch die explizite Typisierung von `f` ergeben sich zwei mögliche Typen für die Variable `o`: `Integer` und `Double`. Als zusätzliche Bedingung gilt hier auch noch, dass `f` in beiden Überladungen `Number` als Rückgabetyp hat. Konkret heißt das, dass es hier für den Case zwei Möglichkeiten gibt: `case R(Integer o)` und `case R(Double o)`. Normalerweise werden solche Fälle durch Überladung auf Funktionsebene gelöst, hier haben jedoch beide Möglichkeiten dieselbe Signatur von `Number m(R r)`. Dadurch fallen beide Methoden zusammen und die Überladung ergibt sich ausschließlich aus einem neu eingefügten Case innerhalb derselben Funktion.

## 4 Pattern Matching in Funktionsköpfen

Wie bereits in Listing 1 zu sehen, kann in Haskell ein Pattern auf Funktionsebene definiert werden. Es gibt auch eine Case-Expression, die ähnlich zur Switch-Expression in Java aufgebaut ist. Im Normalfall wird eine Funktion jedoch durch die einzelnen Fälle im Definitionsbereich definiert.

In Java-TX soll es möglich sein, im Funktionskopf anstatt von normalen Parametern auch Patterns zu verwenden. Im Grunde genommen sind Funktionsparameter ohnehin eine spezielle Form von Pattern, die sich als Typ-Pattern in Switch-Expressions wiederfinden.

Im einfachsten Fall ändert sich für die Klassendefinition nichts, und es wird lediglich eine neue Funktion definiert, welche in einer Präambel die Dekonstruktion der Parameter übernimmt. Der Rest der Funktion sieht nur die neu definierten Variablen.

```
record R(int x, int y) {}

// int m(R r)
m(R(x, y)) {
  return x + y;
}
// int m(int x)
m(x) {
  return x * 2;
}
```

**Listing 7:** Pattern mit eindeutiger Überladung

Hier können problemlos mehrere Methoden des gleichen Namens definiert werden, da die Signaturen jeweils eindeutig sind. Wenn von anderer Stelle `m` referenziert wird, kann die Typinferenz die korrekte Methode auswählen.

Interessanter wird es jedoch, wenn Methoden definiert werden, die keine eindeutige Unterscheidung der Methoden zur Kompilierzeit ermöglichen. Das passiert, wenn geschachtelte Patterns verwendet werden.

```
record R(Number x) {}

// int m(R r)
m(R(int x)) {
  return x;
}
m(R(double x)) {
  return (int) (x + 1);
}
```

**Listing 8:** Pattern mit uneindeutiger Überladung

Diese zwei Methoden machen nur als Einheit Sinn. Zur Laufzeit muss entschieden werden, welche konkrete Funktion aufgerufen werden soll. Im Prinzip muss hier eine gemeinsame Funktion generiert werden, welche aus einer Switch-Expression besteht, die jeweils die konkrete Implementierung aufruft.

```
int m$1(R r) {
  // Praeambel
  var x = (Integer) r.x;
  // Funktionskoerper
  return x;
}
int m$2(R r) {
  var x = (Double) r.x;
  return (int) (x + 1);
}

int m(R r) {
  return switch(r) {
    case R(int x)    -> m$1(r);
    case R(double x) -> m$2(r);
    default -> throw ...
  };
}
```

**Listing 9:** Generierte Methoden

Hier kann die schon vorhandene Implementierung von Switch-Expressions verwendet werden. Die zwei originalen Methoden müssen umbenannt werden, damit die Signatur eindeutig wird. Von außen und aus Sicht der Typinferenz existiert nur die generierte Methode. Alle weiteren Aufrufe von `m` werden also korrekt abgebildet.

## 5 Beispiel: Konkatenation von Listen

In diesem Beispiel wird ein einfacher Algorithmus zur Konkatenation von Listen vorgestellt. Wir orientieren uns wieder an Haskell und zeigen dann, wie das Beispiel in Java-TX aussehen könnte.

```
data List a = [] | a : List a

append :: [a] -> [a] -> [a]
append [] ys = ys
append (x:xs) ys = x : (append xs ys)
```

**Listing 10:** Konkatenation in Haskell

Listen sind in Haskell schon vordefiniert, deshalb ist diese Datendefinition als Pseudocode zu verstehen.

Um zwei (verknüpfte) Listen zu konkatenieren, gibt es hier eine Fallunterscheidung. Der erste Fall entspricht der leeren Liste konkateniert mit einer beliebigen Liste. Hier wird einfach die zweite Liste zurückgegeben. Dieser Fall bricht die Rekursion ab.

Der zweite Fall nimmt das erste Element einer nicht leeren Liste und fügt dieses am Anfang einer neuen Liste an. Der Rest dieser Liste entspricht dem rekursiven Aufruf von `append` mit allen weiteren Elementen der ersten Liste als erstes Argument, und der zweiten Liste als weiteres Argument.

Um diesen Algorithmus in Java-TX abzubilden, muss zunächst die Datendefinition übernommen werden. Das geschieht analog zu Unterabschnitt 2.4. Die beiden Wertkonstruktoren werden als Records `Empty` für die leere Liste und `Cons` für die einzelnen Glieder der Liste implementiert. Der übergeordnete Typ ist das Sealed Interface `List`.

```
sealed interface List<T>
  permits Cons, Empty {}

record Cons<T>(T a, List<T> l)
  implements List<T> {}

record Empty<T>()
  implements List<T> {}
```

**Listing 11:** Datendefiniton für List

Die Implementierung von Append folgt ebenfalls dem Beispiel von Haskell. Es werden zwei Methoden erstellt, die jeweils eine der Möglichkeiten abdecken.

```
append(Cons(a, b), list2) {
  return new Cons<>(
    a, append(b, list2));
}

append(Empty(), list2) {
  return list2;
}
```

**Listing 12:** Append Methode in Java-TX

Als ersten naiven Ansatz könnte man diese Methoden ähnlich wie in Listing 9 zusammenfassen. Diese Variante wäre vermutlich die naheliegendste bei einer manuellen Übertragung. Sie ignoriert allerdings das Ergebnis der Typinferenz, dazu mehr in Unterabschnitt 5.1.

```
<T> Cons<T> append$1(
  Cons<T> c, List<T> list2) {
  return new Cons<>(c.a(),
    append(c.l(), list2));
}

<T> List<T> append$2(
  Empty<T> e, List<T> list2) {
  return list2;
}

<T> List<T> append(
  List<T> a, List<T> b) {
```

```
  return switch(a) {
    case Cons<T> c -> append$1(c, b);
    case Empty<T> e -> append$2(e, b);
  };
}
```

**Listing 13:** Übersetzung Append Methode

Diese Überlegung basiert darauf, dass beide Methoden Teil einer gemeinsamen Methode darstellen. Das ist in Haskell auch der Fall, da Funktionen in Haskell nicht überladen werden können. In dieser zusammengefassten Methode kann jetzt ein allgemeinerer Typ `List` als Argument angenommen werden. Für die ursprünglichen Methoden passt `List` auch als zweiter Parameter. Das Ergebnis ist insofern perfekt, weil die Intention des Programmierers umgesetzt wurde. Gewollt wurde eine Methode, die zwei Listen konkateniert. Genau das ist das Ergebnis.

### 5.1 Ergebnis der Typinferenz

Listing 13 mag zwar wie das richtige Ergebnis erscheinen, allerdings ist es von der falschen Richtung her gedacht. Sie ignoriert die logische Abfolge der Schritte des Compilers, nämlich:

Parsen – Constraints generieren – Typinferenz – AST Transformation – Codegenerierung

Jeder dieser Schritte wird nur einmal ausgeführt. Um unser „perfektes" Ergebnis zu bekommen, müsste die Typinferenz wissen, dass beide Append-Methoden zusammengehören. Dies können wir leider während der Constraintgenerierung nicht herausfinden, weil die Typen zu diesem Zeitpunkt noch nicht berechnet sind. Konkret ist zwar der erste Parameter eindeutig, da das Pattern den Typ direkt angibt. Der zweite Parameter allerdings hat noch keinen festen Typ angenommen. Die einzige Möglichkeit wäre also, die Typinferenz so abzuändern, dass hier Methoden zusammengefasst werden können. Das würde allerdings die Typinferenz komplizierter machen und wäre nur unter großem Aufwand umsetzbar.

Unser Ansatz ist demnach, die Typinferenz ohne Änderungen laufen zu lassen und dann den AST so zu transformieren, dass das gewünschte Ergebnis erreicht wird.

Nach dem Parsen werden an allen Stellen, wo Typen eingesetzt werden sollen, sogenannte Typplatzhalter eingefügt. Die Typinferenz liefert eine Liste von Ergebnismengen. Jede dieser Ergebnismengen stellt eine korrekte Typisierung des Programms dar. In der Ergebnismenge stehen die konkreten Typen für die Typplatzhalter und übrig gebliebene Bedingungen, welche in einem späteren Schritt in Generics umgewandelt werden. Die Ergebnismenge unterscheidet nicht nach Funktionen, sie enthält immer das gesamte Programm. Es kann also durchaus passieren, dass einerseits mehrere Methoden in der Signatur dieselbe Typisierung bekommen, sie sich andererseits aber im Inhalt unterscheiden. Was in einem solchen Fall genau passieren soll, ist noch nicht vollständig definiert. Die Überlegungen aus Abschnitt 3 betreffen allerdings genau so einen Fall.

Für unser Beispiel liefert die Typinferenz drei Ergebnismengen. Eingesetzt in die Methodenköpfe sehen die Signaturen folgendermaßen aus:

```
1. Methode mit Cons:
Cons append(Cons(Cons a, Empty b), Empty list2)
Cons append(Cons(Cons a, Empty b), Cons list2)
<AU> Cons append(Cons(Cons a, Cons b), AU list2)

2. Methode mit Empty:
Empty append(Empty(), Empty list2)
Cons append(Empty(), Cons list2)
<BG> BG append(Empty(), BG list2)
```

Eine interessante Beobachtung ist hier, dass die geschachtelten Patterns mit `Cons` hier jeweils eigene konkrete Typen bekommen. Wie sich später herausstellen wird, ist das der springende Punkt, der es möglich macht, die Rekursion zu terminieren.

Von uns wird nun folgender Ansatz verfolgt: *Alle Fälle, die durch Javas Überladungsmechanismus nicht mehr unterscheidbar sind, werden zusammengefasst.*

Wir betrachten jeweils die erste Variante von `append`. Das erste Argument fällt in jedem Fall zusammen, hier passiert also nichts Besonderes. Interessant ist der zweite Parameter. Der erste und der zweite Fall sind unterscheidbar, da `Cons` und `Empty` verschiedene Typen sind. Der erste und dritte Fall allerdings sind nicht unterscheidbar, da für das Generic `AU` auch der Typ `Empty`

eingesetzt werden kann. Genauso verhält es sich mit dem zweiten und dritten Fall.

Analog dazu werden auch die Fälle der zweiten Methode zusammengefasst. Die Zusammenfassung erfolgt auf der Ebene der Ergebnismenge. Das heißt, falls zwei Fälle der ersten Methode zusammenfallen, passiert das auch mit der zweiten Methode. In diesem Fall funktioniert das auch korrekt, da die Signaturen kompatibel sind.

Insgesamt entstehen also vier Methoden, welche die Kombinationen aus `Cons` und `Empty` abdecken. Die ursprünglichen acht Varianten werden beibehalten und umbenannt. Wichtig hier ist, dass für den zweiten Parameter gematcht werden muss. Für `AU` wird beispielsweise `Cons` eingesetzt. Das ist wichtig, damit im Funktionskörper die richtige Variante von `append` aufgerufen wird. Nur so entsteht korrekter Code.

Das Ergebnis ist im Folgenden stark vereinfacht dargestellt. Die ursprünglichen Methoden werden direkt in das Ergebnis eingefügt, anstatt umbenannt und im generierten Code aufgerufen. Überflüssige Switch-Expressions wurden entfernt.

```
Cons append(Cons l1, Cons l2) {
  return switch(l1) {
    case Cons(var a, Cons b)
      -> new Cons(a, append(b, l2));
    case Cons(var a, Empty b)
      -> new Cons(a, append(b, l2));
  }
}
Cons append(Cons l1, Empty l2) {
  return switch(l1) {
    case Cons(var a, Cons b)
      -> new Cons(a, append(b, l2));
    case Cons(var a, Empty b)
      -> new Cons(a, append(b, l2));
  }
}

Cons append(Empty l1, Cons l2) {
  return l2;
}
Empty append(Empty l1, Empty l2) {
  return l2;
}
```

**Listing 14:** Ergebnis Append Methode

Durch die Unterscheidung in der Struktur von `Cons` wird bei den ersten zwei Append-Methoden jeweils eine andere Methode aufgerufen. Der erste Case ruft die eigene Methode auf, während der zweite Case in eine der unteren Append-Methoden springt, die das zweite Argument ohne Änderungen zurückgibt. So wird die Rekursion abgebrochen und das Programm verhält sich korrekt.

Ein paar Fragen bleiben noch offen. Zunächst werden hier Rawtypes verwendet, das heißt, die generischen Parameter wurden weggelassen. Ob und wie diese automatisch generiert werden können, ist noch nicht klar. Ein weiteres Problem stellt dar, dass unser Ergebnis aus Listing 13 nicht ganz erreicht wird. Es werden zwar alle Fälle abgedeckt, aber normalerweise, wenn mit Listen gearbeitet wird, ist die Eingabe auch vom Typ `List` und nicht `Cons`. Dieses Problem lässt sich durch weiteres Zusammenfassen lösen. Es könnte eine Bridge-Methode erstellt werden, die je nach Kombination der Parameter die richtige konkrete Funktion aufruft.

Interessant wäre ein Fall bei dem die Typinferenz den allgemeineren Typ `List` einsetzt, weil dieser an anderer Stelle benötigt wird.

## 6 Zusammenfassung und Ausblick

In diesem Paper wurde Pattern Matching als Konzept vorgestellt. Ansätze aus Java und Haskell wurden verglichen und Konzepte aus beiden Programmiersprachen in Java-TX integriert. Aus Haskell übernommen und angepasst wurde das Pattern Matching in Funktionsköpfen, welches teilweise durch Methodenüberladung und Bridge-Methoden umgesetzt worden ist. Mit der Konkatenation von Listen wurde ein komplexes Beispiel vorgestellt, anhand dessen exemplarisch eine Umsetzungsstrategie für Java-TX beschrieben wurde.

In einer weiteren Arbeit müssen die theoretischen Grundlagen erarbeitet werden, um einen Algorithmus zu finden, der in jedem Fall das korrekte Ergebnis liefert. Komplexere Beispiele verhalten sich möglicherweise anders als hier beschrieben.

## Quellen


[1] Gavin Bierman. *JEP 361: Switch Expressions*. Mar. 11, 2022. URL: https://openjdk.org/jeps/361.

[2] Gavin Bierman. *JEP 395: Records*. Feb. 3, 2024. URL: https://openjdk.org/jeps/395.

[3] Gavin Bierman. *JEP 409: Sealed Classes*. Jan. 3, 2024. URL: https://openjdk.org/jeps/409.

[4] Gavin Bierman. *JEP 440: Record Patterns*. Aug. 28, 2023. URL: https://openjdk.org/jeps/440.

[5] Ian Darwin. *The Visitor Design Pattern in Depth*. July 15, 2019. URL: https://blogs.oracle.com/javamagazine/post/the-visitor-design-pattern-in-depth.

# Lemming: A Novel Runtime System for Cyber-physical Systems

Nils Scheidweiler and Clemens Grelck
Friedrich Schiller University Jena
{nils.scheidweiler,clemens.grelck}@uni-jena.de

### Abstract

Cyber-physical systems (CPSs) do not only require functional correctness, but also must comply with non-functional properties, such as energy, time, security, and robustness (ETSR). Deploying CPSs on parallel and heterogeneous computing platforms introduces complex scheduling and mapping challenges to meet ETSR constraints and objectives. We introduce Lemming, our novel runtime system designed to address these challenges. It adopts the TeamPlay coordination model: applications are organized as directed acyclic graphs (DAGs) consisting of tasks that implement user-defined computations. Edges connect tasks and define dependencies and stream-based data exchange. Lemming serves as a highly configurable toolbox. Therefore, it can be used for a wide range of applications with different ETSR properties. We validate Lemming's efficacy and efficiency through a case study that implements a digital signal processing application from the music domain.

## 1 Introduction

A cyber-physical system (CPS) integrates computational and physical capabilities [4]. Sensors and actuators serve as interfaces with the physical world [10]. The cyber-physical system receives input from the sensors, performs computations, and controls the actuators. These systems are prevalent in our world with applications ranging from medical devices to aerospace systems [17, 13].

Non-functional properties such as energy, time, security, and robustness (ETSR) are as important for CPSs as functional correctness [2]. The ETSR properties are illustrated in Figure 1. For battery-powered CPSs the energy consumption is important [7, 9]. A CPS must typically respond to sensor inputs with actuator output before some deadline and, therefore, has real-time constraints [18]. Security is also important, as a CPS can have a harmful or even disastrous impact on the physical environment [11]. As CPSs are often connected to a network infrastructure, they can become targets of cyber attacks [8, 12]. In addition, the robustness against software and hardware faults can also be important [14, 16].

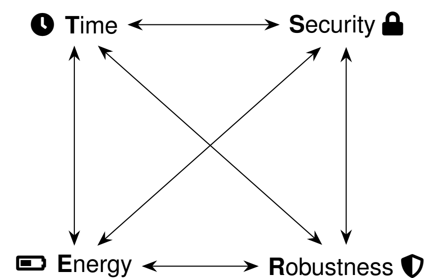


**Figure 1:** Non-functional properties and their interdependence

The arrows in Figure 1 emphasize the interdependence of non-functional properties. For instance, dynamic voltage and frequency scaling (DVFS) [5, 21] can be used to reduce energy consumption, but increases computation time. N-modular redundancy is used to increase fault-tolerance at the cost of higher energy usage be-

cause the same computation is executed multiple times. This can be done in parallel, provided that enough computation units are available at that time. Otherwise, redundancy affects the ability to meet deadlines. Similarly, increasing security by strengthening, for example, the encryption of wireless communication increases energy consumption and requires more time. A CPS must balance these properties on the basis of the constraints and objectives specific to the application.

Modern computing systems are parallel and heterogeneous. These have multiple CPUs, GPUs, and other specialized computation units. In addition to the ETSR constraints that must be met, a CPS must also achieve various objectives. One common objective is to minimize energy consumption [18]. The decision on which computation unit and when to run a job on such systems in order to achieve its objectives presents a challenging problem. Solving this directly in each and every CPS application causes additional effort.

We present our novel runtime system *Lemming* [20] to address these challenges. Lemming facilitates the deployment of CPS applications on parallel and heterogeneous computing platforms while taking ETSR constraints and objectives into account. Our runtime system serves as a middleware between the user application and the operating system. It separates coordination concerns from application programming, allowing users to focus on implementing application-specific code while Lemming handles scheduling, mapping, and inter-task communication. Lemming is designed as a library for Linux and serves as a toolbox the user can select the runtime system's features from. This includes, for example, application-specific weighting of ETSR constraints and objectives.

We demonstrate the efficacy and efficiency of Lemming using a case study that implements a DSP (digital signal processing) application from the music domain.

This paper is organized as follows: In Section 2 we present the system model used for our runtime system. In Section 3 we describe the design and implementation of Lemming. In Section 4 we present our case study from the music domain. Finally, in Section 5 we draw conclusions from our paper.

## 2 System model

Lemming adopts the *TeamPlay coordination model* [18]. Gelernter and Carriero introduce the term coordination with their coordination language Linda [6]. Coordination code which is integrated within the computation code is called endogenous. Otherwise, coordination code which separates coordination and computation is called exogenous [3]. TeamPlay is an exogenous approach that separates coordination concerns from application code. Applications in TeamPlay are organized as directed acyclic graphs (DAGs), where tasks (vertices) implement user-defined operations, and edges called channels define timing dependencies and stream-based communication. Tasks communicate via typed inports and outports. The data items that are transmitted via channels are called tokens. A source task is a special task that lacks inports and is used as an interface to one or many sensors. In the same way, a sink task lacks outports and is used as interface to one or many actuators. Tasks are stateless, but a state can be modeled using short-circuited channels from an outport to an inport of the same tasks. DAG execution is periodic and must complete before a deadline [18].

Each task has a worst-case execution time (WCET) and a worst-case energy consumption (WCEC), which depend on the target platform. These can be derived through static analysis or profiling [1, 22]. TeamPlay also supports multi-version tasks, allowing selection of optimal implementations based on ETSR objectives. For example, a task that performs asymmetric encryption can implement different security levels in its task versions. The higher the security level of a task, for example, by increasing the RSA key length, the more energy and time the encryption requires. If the objective is to reduce the energy consumption, Lemming will prefer task versions with a lower security level.

In Figure 2 we show an example of a TeamPlay DAG that we use as a running example in this paper. Task 1 which is the source task executes every 20 ms, triggering Task 2 and Task 3. When the execution of Task 2 and Task 3 is completed, Task 4 executes.

Applications are specified using the TeamPlay coordination language. A compiler from the TeamPlay toolchain generates an executable from the code written in TeamPlay [18].

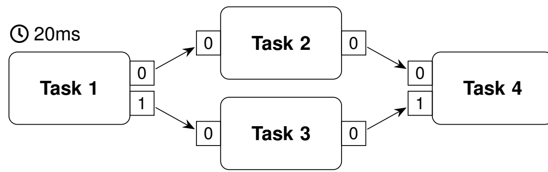


**Figure 2:** A TeamPlay DAG with four tasks

## 3 Lemming

In this section we describe our runtime system Lemming. It is developed in `C` and designed as a library. The user describes the application with the Lemming configuration specification using calls to the Lemming API. Lemming creates the Lemming configuration from the specification that is internally used as a central data structure to describe the user application.

### 3.1 Architecture

Lemming consists of several components whose interplay is illustrated in Figure 3.

- **Controller**: The controller is the central orchestrator and, hence, started first. It creates a new process for each subcontroller. Furthermore, it creates a UNIX socket and shared memory for communication between the runtime system components.
- **Subcontrollers**: The subcontrollers manage a group of tasks and a subset of the system's computation units and orchestrate the task execution on these computation units. The subcontrollers start a worker for each computation unit. Furthermore, the subcontrollers spawn a new process or thread for each task.
- **Workers**: Each worker controls a computation unit. The workers manage the execution of tasks on its computation units.
- **Tasks**: Tasks serve as interface between the runtime system and user code. They handle the communication with other tasks and execute the user code. Each task's inport and outport has a unique index.
- **Plugins**: The user code is implemented in plugins. These can be defined directly in the application code or loaded from shared object files. The user code reads from the inports, executes computations, and writes the results to the task's outports.

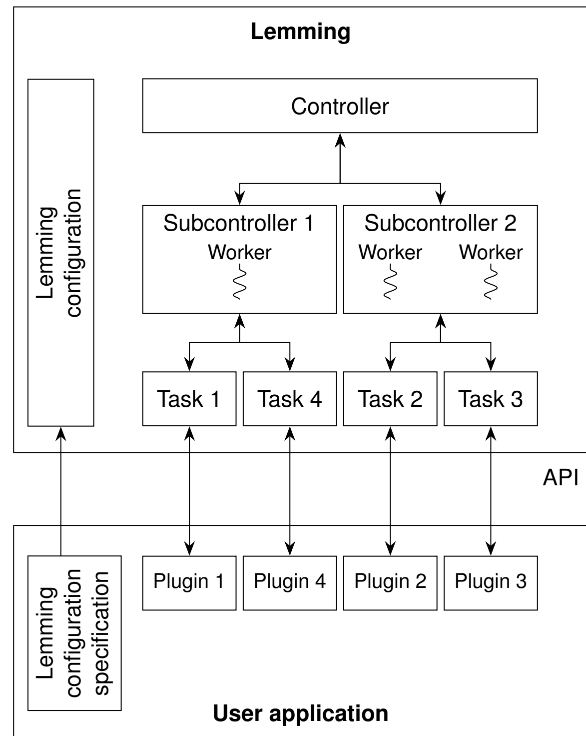


**Figure 3:** Lemming architecture overview

- **Channels**: Each channel connects a task's outport with an inport. In addition to the default channels which only save one token, Lemming offers stream channels and persistent channels. Stream channels can store multiple tokens. Persistent channels store a token until it is overwritten by writing a new token to the channel.

Figure 3 shows an overview of the Lemming components for the running example DAG introduced in Figure 2. In this example Task 1 and Task 4 are assigned to Subcontroller 1 and Task 2 and Task 3 to Subcontroller 2. Subcontroller 1 controls one computation unit and Subcontroller 2 controls two computation units of the system.

### 3.2 Task scheduling

Lemming offers three task scheduling classes: offline scheduling, global online scheduling, and partitioned online scheduling. Task scheduling is done non-preemptively. Task migration is allowed when offline or global online scheduling is used.

With offline scheduling, the user supplies a schedule table to the runtime system via the Lemming configuration. It specifies the release time and the target computation unit for each job. When

```c
struct lem_config config;
lem_config_init(&config);

// Task configuration
struct lem_task_attr task_attr;
lem_task_attr_init(&task_attr);
lem_task_attr_set_name(&task_attr, "Task 1");
lem_task_attr_set_plugin_path(&task_attr,
    "libplugin1.so");
lem_task_attr_set_rule(&task_attr,
    LEM_TASK_RULE_PERIODIC);
lem_task_attr_set_period(&task_attr, ms(10));
lem_task_attr_set_wcet(&task_attr, ms(1));
lem_task_attr_set_inport_count(&task_attr, 0);
lem_task_attr_set_outport_count(&task_attr, 2);
lem_task_attr_set_process(&task_attr, true);
struct lem_task_config *task1 =
    lem_config_add_task(&config, 1, &task_attr);
```

**Figure 4:** Add Task 1 from the running example to the Lemming configuration

the user selects online scheduling, the scheduling decisions are made during runtime. In the case of global online scheduling, each task can be executed on every computation unit of the task's associated subcontroller. Otherwise, in the case of partitioned online scheduling, a task can only execute on its assigned computation unit. For online scheduling, a schedulability analysis must be performed beforehand.

### 3.3 Process scheduling

Each task is executed in its own process or thread. Lemming controls the process scheduling and mapping of the operating system to comply with real-time constraints and to enforce the task scheduling decisions. For this purpose, every process and thread of the runtime system is executed using the real-time-capable `SCHED_FIFO` scheduling policy and the process priority is set to maximum. In order to avoid interference with other non-runtime system processes, the user has to isolate the CPU cores that are assigned to the subcontrollers. This can be done using the boot parameter `isolcpus`, which modifies the default process affinity mask to exclude the specified CPU cores [15]. Additionally, users must modify the Linux utilization limit for real-time processes, as the default limit is set to 95%. This can be overridden by changing the `sched_rt_runtime_us` parameter [19].

### 3.4 Lemming configuration specification

Figure 4 illustrates how to add a task to the Lemming configuration using the Lemming API. The configuration is stored in a variable of the type `lem_config`. To add a task, the user specifies task attributes, which include the task name, the path to the plugin file, and the execution rule that determines whether the task runs periodically, is triggered by predecessor tasks, or runs sporadically or aperiodically. For periodic tasks, the attributes are used to set the execution period. Additionally, the task attributes are used to set the worst-case execution time (WCET), and specify the number of inports and outports. Furthermore, task attributes are used to set whether the task is executed in a process or thread. The task is added to the Lemming configuration using the `lem_config_add_task` function. This function takes three arguments: the target Lemming configuration, a unique task ID, and the task attributes. It returns a pointer to the task for future reference.

### 3.5 Plugins

In Figure 5, we show an example of a plugin that performs a string reverse. The function `plugin_initialize` is called by our runtime system and sets the run handler function, which is called on every task execution. In the run function tokens are retrieved from the task's inports using `lem_task_get`. The port index is passed to `lem_task_get` as an argument. In the same way, tokens can be written to an outport using `lem_task_put`. In the example, tokens of the data type `type` are received and sent.

## 4 Case study

To validate the efficacy and efficiency of Lemming, we implemented a digital signal processing (DSP) application from the music domain as a case study. This application processes audio from multiple instrument inputs, applies effects, and plays the mixed output via loudspeakers. The DSP application interacts with the physical world through audio inputs from musical instruments and audio outputs to loudspeakers. It has real-time requirements because the audio processing must meet a timing deadline.

```c
int plugin_initialize(struct lem_plugin_context
    *plugin_ctx) {
    lem_registry_set_run_handler(
        plugin_ctx->registry, plugin_run);
    return 0;
}

void plugin_run(struct lem_plugin_context
    *plugin_ctx) {
    struct type *in = (struct type *)
        lem_task_get(plugin_ctx->task, 0);
    struct type *out = (struct type *)
        lem_task_put(plugin_ctx->task, 0);

    char *in_str = in->str;
    char *out_str = out->str;

    size_t length = strlen(in_str);
    for (size_t i = 0; i < length; i++) {
        out_str[i] = in_str[length - 1 - i];
    }
    out_str[length] = '\0';
}
```

**Figure 5:** Plugin example for a task that performs a string reverse

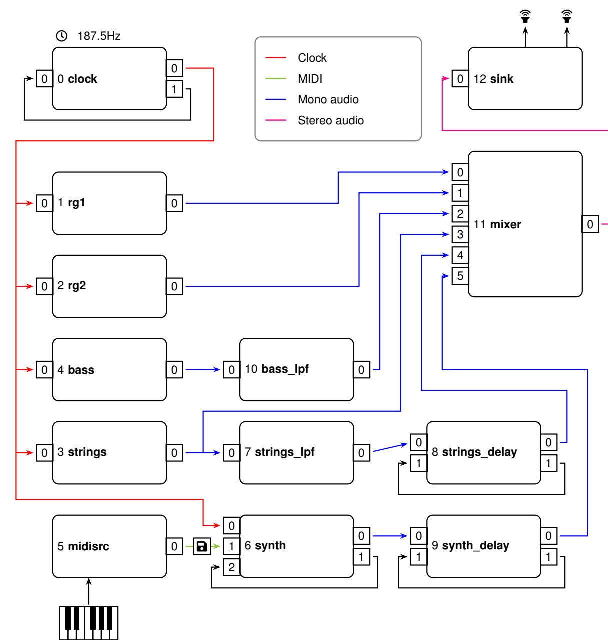


**Figure 6:** DAG of the audio DSP application

The continuous audio signal of an instrument is sampled and quantized into a discrete input buffer. The DSP application applies effects and outputs the processed signal to an output buffer, which is then transferred to the sound card or audio interface for playback.

The DSP application is modeled as a DAG, as shown in Figure 6. The `clock` task serves as a source task. It is executed periodically to initiate audio signal processing. The tasks `rg1` (rhythm guitar 1), `rg2` (rhythm guitar 2), `bass` (bass guitar), and `strings` stream audio from memory to simulate live-played instruments. The `midisrc` task (MIDI source) is a sporadic task triggered by events from a MIDI device, such as an external keyboard. The MIDI source task writes the MIDI data to a persistent channel, which is marked with a memory icon in the figure. The `synth` task (synthesizer) generates a mono-audio signal from MIDI data. Because the MIDI data is stored in a persistent channel, the synthesizer can access the current state of the pressed keys each time the task is executed. We apply effects such as low-pass filters and delays to the audio signals. The `mixer` task combines the incoming mono-audio signals and outputs a stereo-audio signal, which is played through loudspeakers by the sink task.

## 5 Conclusion

Lemming is a novel runtime system designed to address the challenges of deploying CPSs on parallel and heterogeneous computing platforms such as real-time scheduling, mapping, inter-task communication, and meeting non-functional objectives and constraints. By adopting the Team-Play coordination model, Lemming separates coordination concerns from application programming, allowing users to focus on implementing application-specific code. Therefore, our runtime system speeds up the development time of CPS applications. Lemming increases the confidence in the correctness of the solution, and last but not least, lowers the bar for the qualification of system engineers. Lemming acts as a highly configurable toolbox from which the user can select the best combination of features. This makes Lemming suitable for a wide range of applications. The efficacy and efficiency of Lemming are demonstrated through a case study implementing a digital signal processing application from the music domain.

As future work, we plan to implement task isolation and fault-tolerance techniques for security and robustness.

## References


[1] Jaume Abella et al. "WCET analysis methods: Pitfalls and challenges on their trustworthiness". In: *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2015, pp. 1–10.

[2] Benny Akesson et al. "An Empirical Survey-based Study into Industry Practice in Real-time Systems". In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. 2020, pp. 3–11.

[3] Farhad Arbab. "Composition of Interacting Computations". In: *Interactive Computation: The New Paradigm* (Jan. 2006).

[4] Radhakisan Baheti and Helen Gill. "Cyber-physical systems". In: *The impact of control technology* 12.1 (2011), pp. 161–166.

[5] Mario Bambagini et al. "Energy-Aware Scheduling for Real-Time Systems: A Survey". In: *ACM Trans. Embed. Comput. Syst.* 15.1 (Jan. 2016). ISSN: 1539-9087. URL: https://doi.org/10.1145/2808231.

[6] David Gelernter and Nicholas Carriero. "Coordination languages and their significance". In: *Commun. ACM* 35.2 (Feb. 1992), pp. 97–107. ISSN: 0001-0782. URL: https://doi.org/10.1145/129630.129635.

[7] Marco E. Gerards, Johann L. Hurink, and Philip K. Hölzenspies. "A survey of offline algorithms for energy minimization under deadline constraints". In: *J. of Scheduling* 19.1 (Feb. 2016), pp. 3–19. ISSN: 1094-6136. URL: https://doi.org/10.1007/s10951-015-0463-8.

[8] Houda Harkat et al. "Cyber-physical systems security: A systematic review". In: *Computers & Industrial Engineering* 188 (2024), p. 109891. ISSN: 0360-8352. URL: https://www.sciencedirect.com/science/article/pii/S0360835224000123.

[9] Houssam Kanso, Adel Noureddine, and Ernesto Exposito. "A Review of Energy Aware Cyber-Physical Systems". In: *Cyber-Physical Systems* 10.1 (2024), pp. 1–42. eprint: https://doi.org/10.1080/23335777.2022.2163298. URL: https://doi.org/10.1080/23335777.2022.2163298.

[10] S.K. Khaitan and James Mccalley. "Design Techniques and Applications of Cyberphysical Systems: A Survey". In: *IEEE Systems Journal* 9 (July 2014), pp. 1–16.

[11] Kyoung-Dae Kim and Panganamala Kumar. "Cyber–Physical Systems: A Perspective at the Centennial". In: *Proceedings of The IEEE - PIEEE* 100 (May 2012), pp. 1287–1308.

[12] Charalambos Konstantinou et al. "Cyber-physical systems: A security perspective". In: *Proceedings - 2015 20th IEEE European Test Symposium, ETS 2014* (June 2015).

[13] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. 2nd. The MIT Press, 2016. ISBN: 0262533812.

[14] Wouter Loeve and Clemens Grelck. "Fault-tolerance at your Finger Tips with the TeamPlay Coordination Language". In: *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*. IFL '23. Braga, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400716317. URL: https://doi.org/10.1145/3652561.3652571.

[15] Michael M. Madden. "Challenges using Linux as a real-time operating system". In: *AIAA Software Challenges in Aerospace* 2019-0502 (Jan. 2019). URL: https://doi.org/10.2514/6.2019-0502.

[16] Risat Mahmud Pathan. "Fault-tolerant and real-time scheduling for mixed-criticality systems". In: *Real-Time Syst.* 50.4 (July 2014), pp. 509–547. ISSN: 0922-6443. URL: https://doi.org/10.1007/s11241-014-9202-z.

[17] Ragunathan Rajkumar et al. "Cyber-physical systems: The next computing revolution". In: *Design Automation Conference*. 2010, pp. 731–736.

[18] Julius Roeder et al. "Towards Energy-, Time- and Security-Aware Multi-core Coordination". In: *Lecture Notes in Computer Science*. Ed. by Simon Bliudze and Laura Bocchi. Vol. LNCS-12134. Coordination Models and Languages. Part 2: Coordination Languages. Valletta, Malta: Springer International Publishing, June 2020, pp. 57–74. URL: https://inria.hal.science/hal-03273984.

[19] Benjamin Rouxel et al. "PReGO: a generative methodology for satisfying real-time requirements on COTS-based systems: Definition and experience report". In: *19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2020), Chicago, USA*. ACM, 2020, pp. 70–83. URL: https://www.

lexuor.net/publications/Drone_Use_Case_preprint.pdf.

[20] Nils Scheidweiler. "Lemming: Design and Implementation of a Novel Runtime System for Cyber-physical Systems". Friedrich Schiller University Jena. Master's thesis. Aug. 2025.

[21] Dawei Sun et al. "Re-Stream: Real-time and energy-efficient resource scheduling in big data stream computing environments". In: *Information Sciences* 319 (2015), pp. 92–112. ISSN: 0020-0255. URL: https://www.sciencedirect.com/science/article/pii/S0020025515001929.

[22] Peter Wägemann et al. "Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems". In: *2015 27th Euromicro Conference on Real-Time Systems*. 2015, pp. 105–114.

# Type Inference for Java using ASP – First approach

Andreas Stadelmeier
Baden-Wuerttemberg Cooperative State University
Horb, Germany
andi@paulus.haus

## Abstract

Global type inference for Java is able to compute correct types for an input program which has no type annotations at all, but turns out to be NP-hard. Former implementations of the algorithm therefore struggled with bad runtime performance. In this paper we translate the type inference algorithm for Featherweight Generic Java to an Answer Set Program. Answer Set Programming (ASP) promises to solve complex computational search problems as long as they are finite domain. This paper shows that it is possible to implement global type inference for Featherweight Generic Java (FGJ) as an ASP program. Afterwards we compared the ASP implementation with our own implementation in Java exposing promising results. Another advantage is that the specification of the algorithm can be translated one on one to ASP code leading to less errors in the respective implementation. Unfortunately ASP can only be used for type inference for a subset of the Java type system that excludes wildcard types.

## 1. Global Type Inference

Java is a strictly typed programming language. The current versions come with a local type inference feature that allows the programmer to omit type annotations in some places like argument types of lambda expressions for example. But methods still have to be fully typed including argument and return types. We work at a global type inference algorithm for Java which is able to compute correct types for a Java program with no type annotations at all.

A possible input for our algorithm is shonw in figure 1. Here the method `untypedMethod` is missing its argument type for `var` and its return type. Global type inference works in two steps. First we generate a set of subtype constraints, which are later unified resulting in a type solution consisting of type unifiers $\sigma$. Every missing type is replaced by a type placeholder. In this example the type placeholder $var$ is a placeholder for the type of the `var` variable. The $ret$ placeholder represents the return type of the `untypedMethod` method. Also shown as a comment behind the respective method call. There are two types of type constraints. A subtype constraint like $var \lessdot \texttt{C1}$ meaning that the unification algorithm has to find a type replacement for $var$ which is a subtype of `C1`. Due to the Java type system being reflexive one possible solution would be $\sigma(var) = \texttt{C1}$. A constraint $var \doteq \texttt{C1}$ directly results in $\sigma(ret) = \texttt{C1}$.

Our type inference algorithms for Java are described in a formal manner in [3], [6]. The first prototype implementation put the focus on a correct implementation rather than fast execution times. Depending on the input it currently takes upto several minutes or even days to compute all or even one possible type solution. To make them usable in practice we now focus on decreasing the runtime to a feasible level.

```
class C1 {
    C1 self(){
        return this;
    }
}
class C2 {
    C2 self(){
        return this;
    }
}
class Example {
    untypedMethod(var){
        return var.self();    //   ⟹ {var ⋖ C1, ret ≐ C1} || {var ⋖ C2, ret ≐ C2}
    }
}
```

**Figure 1:** Java program missing type annotations

## 2. Motivation

The nature of the global type inference algorithm causes the search space of the unification algorithm to increase exponentially with every ambiguous method call. Java allows overloaded methods causing our type inference algorithm to create so called Or-Constraints. This happens if multiple classes implement a method with the same name and the same amount of parameters. Given the input program in figure 1 we do not know, which method `self` is meant for the method call `var.self()`, because there is no type annotation for `var`.

An Or-Constraint like $\{var \lessdot \texttt{C1}, ret \doteq \texttt{C1}\} \,||\, \{var \lessdot \texttt{C2}, ret \doteq \texttt{C2}\}$ means that either the the constraint set $\{var \lessdot \texttt{C1}, ret \doteq \texttt{C1}\}$ or $\{var \lessdot \texttt{C2}, ret \doteq \texttt{C2}\}$ has to be satisfied. If we set the type of `var` to `C1`, then the return type of the method will be `C1` aswell. If we set it to `C2` then also the return type will be `C2`. There are two possibilities therefore the Or-Constraint.

If we chain multiple overloaded method calls together we end up with multiple Or-Constraints. The type unification algorithm **Unify** only sees the supplied constraints and has to potentially try all combinations of them. This is proven in [6] and is the reason why type inference for Featherweight Generic Java is NP-hard. Let's have a look at the following alternation of our example:

```
untypedMethod(var){
    return var.self().self()
        .self().self();
}
```

**Figure 2:** Stacked ambiguous method calls

Now there are four chained method calls leading to four Or-Constraints:

$$
\begin{gathered}
\{var \lessdot \texttt{C1}, r1 \doteq \texttt{C1}\} \,||\, \{var \lessdot \texttt{C2}, r1 \doteq \texttt{C2}\} \\
\{r1 \lessdot \texttt{C1}, r2 \doteq \texttt{C1}\} \,||\, \{r1 \lessdot \texttt{C2}, r2 \doteq \texttt{C2}\} \\
\{r2 \lessdot \texttt{C1}, r3 \doteq \texttt{C1}\} \,||\, \{r2 \lessdot \texttt{C2}, r3 \doteq \texttt{C2}\} \\
\{r3 \lessdot \texttt{C1}, ret \doteq \texttt{C1}\} \,||\, \{r3 \lessdot \texttt{C2}, ret \doteq \texttt{C2}\}
\end{gathered}
$$

The placeholder $r1$ stands for the return type of the first call to `self` and $r2$ for the return type of the second call and so on. It is obvious that this constraint set only has two solutions. The variable `var` and the return type of the method aswell as all intermediate placeholders $r1 - r3$ get either the type `C1` or `C2`. A first prototype implementation of the **Unify** algorithm simply created the cartesian product of all Or-Constraints, 16 possibilities in this example, and iteratively tried all of them. This leads to a exponential runtime increase with every added overloaded method call. Eventhough the current algorithm is equipped with various optimizations as presented in [5] and [4] there are still runtime issues when dealing with many Or-Constraints.

Our global type inference algorithm should construct type solutions in real time. Then it can effectively used as a Java compiler which can deal with large inputs of untyped Java code. Another

use case scenario is as an editor plugin which helps a Java programmer by enabling him to write untyped Java code and letting our tool insert the missing type statements. For both applications a short execution time is vital.

Answer Set programming promises to solve complex search problems in a highly optimized way. The idea in this paper is to implement the algorithm presented in [6] as an ASP program and see how well it handles our type unification problem.

## 3. ASP Implementation

Our ASP implementation replaces the unification step of the type inference algorithm. So the input is a set of constraints $c$ and the output is a set of unifiers $\sigma$. An ASP program consists of implication rules and facts. Here the facts are the input constraints and the rules are the ones depicted in figure 4. After running the resulting program through an interpreter the output holds the desired $\sigma(a) = \mathtt{T}$ literals.

## 4. Proofs

An ASP solver tries to find the smallest possible set of constraints $C_S$, which satisfies all implication rules given in figure 4 while also containing the input constraints $C$. The constraint set $C_S$ is called a stable model. We now want to prove two key properties. soundness and completeness. Our algorithm is sound if it computes only correct solutions and complete if it finds a solution if there exists one. To proof soundness (1) and completeness (2) we have to show that $C_S$ contains a $\sigma(a) = \mathtt{G}$ for every type placeholder $a$ used in the input constraints $C$ and that this unifier $\sigma$ is correct.

The first thing to show is that every type placeholder ends up in a constraint of the form $a \doteq \mathtt{G}$ that is transformed into an unifier $\sigma(a) = \mathtt{G}$ by the SOLUTION rule ($\mathtt{G}$ must not contain any type placeholders).

So first we proof that every type placeholder ends up in a constraint of the form $a \doteq \mathtt{N}$. Then we show in lemma 4.6 that $a \doteq \mathtt{N} \in C_S$ also means that $a \doteq \mathtt{G} \in C_S$, which leads to lemma 4.1 (Closedness). Closedness means that every type placeholder ends up getting a unifier $\sigma$. Together with the substitution lemma 4.4 the soundness proof is apparent: Every constraint $\mathtt{S} \lessdot \mathtt{T} \in C_S$ implies that $\sigma(\mathtt{S}) \lessdot \sigma(\mathtt{T}) \in C_S$ and therefore $\sigma$ must be a valid solution otherwise the algorithm will *fail* by one of the Fail-Rules at the bottom of figure 4.

With this proven we can then show that all possible unifiers are considered by the algorithm (Completeness). For completeness we prove that every type placeholder gets every possible unifier assigned to it.

**Lemma 4.1** (Closedness) *Every type placeholder gets a unifier: If $a \in C$ and the input $C$ $\mathtt{ok}$ then there either exists a stable model $C_S$ with $\sigma(a) \in C_S$ or fail.*

*Proof.* Due to the Solution-Requirement and the S-Object rule we know that every type placeholder appears on the left side of a constraint of the form $a \doteq \mathtt{N}$. With this two preliminaries and lemma 4.2 we know by lemma 4.6 that this also leads to a $a \doteq \mathtt{G}$ constraint and therefore to $\sigma(a) = \mathtt{G}$ through the Solution rule finishing the proof. □

First we prove in lemma 4.1 that any type placeholder $a$ inside a given constraint set is getting atleast the type `Object` assigned ($\sigma(a) = \mathtt{Object}$). Afterwards we prove well-formedness for any solution (stable model) with the following lemma, which is needed for the Substitution lemma.

**Lemma 4.2** *Stable Model is Well-Formed If $C_S$ is a stable model for an input $C$ with $C$ $\mathtt{ok}$ then $C_S$ $\mathtt{ok}$*

*Proof.* We proof this by showing that no rule adds a not well-formed constraint. Assuming $C$ ok then after applying one of the rules of $\Omega$ we get a $C'$ with $C'$ ok.

Case Subst-L intermediate, because no new type is created, only existing ones used in a new constraint.

Case Subst-R, Subst-Equal, Swap, Reduce, Solution-Gen, Solution-Requirement are intermediate aswell for the same reason.

Case Subst-Param, Subst-Param-Left and Subst-Param-Right create a new type $\mathtt{C}\langle\mathtt{P}_1, \ldots \mathtt{G}, \ldots \mathtt{P}_n\rangle$, which is ok by WF-Type.

Case S-Object is correct, because $a$ ok and `Object` ok by definition.

| | | | |
|---|---|---|---|
| $c$ | ::= | $\mathtt{T} \lessdot \mathtt{T} \mid \mathtt{T} \doteq \mathtt{T}$ | Constraint |
| $\mathtt{T}, \mathtt{S}$ | ::= | $a \mid \mathtt{N}$ | Type placeholder or Type |
| $\mathtt{N}$ | ::= | $\mathtt{C}\text{<}\overline{\mathtt{P}}\text{>}$ | Class Type containing type placeholders |
| $\mathtt{G}$ | ::= | $\mathtt{C}\text{<}\overline{\mathtt{G}}\text{>}$ | Class Type not containing type placeholders |
| $\mathtt{P}$ | ::= | $a \mid \mathtt{G}$ | Well-Formed Parameter |

**Figure 3:** Syntax of types and constraints

Case Reduce generates $\mathtt{P_i} \doteq \mathtt{P'_i}$ constraints. For each $\mathtt{P_i}$ and $\mathtt{P'_i}$ there are two possibilities: $\mathtt{P_i} = a$ or $\mathtt{P_i} = \mathtt{G}$ and both times $\mathtt{P_i}$ $\mathtt{ok}$ by definition.

Case Adapt has the same reasoning as the Reduce case.

Case Super generates a constraint $a \doteq \mathtt{N}_m$ where $a$ $\mathtt{ok}$ by definition and $\mathtt{N}_m$ $\mathtt{ok}$ by premise of Super.

Case Solution-Gen and Solution-Requirement have the same reasoning as Super.

□

Lemma 4.3 is part of the substitution lemma 4.4.

**Lemma 4.3** (Substitution-Step) *If $C_S$ is a stable model containing $C$ with $C$ $\mathtt{ok}$ and $\{a \doteq \mathtt{G}\} \in C_S$ then $[\mathtt{G}/a]C \subseteq C_S$.*

*Proof.* We show this for every well-formed constraint in $C$:

1. $a \doteq \mathtt{G}$: Subst-Equal implies $\mathtt{T} \doteq \mathtt{G}$ finishing the case, because $[\mathtt{T}/a]\mathtt{G} = \mathtt{G}$
2. $\mathtt{S} \doteq \mathtt{C}\text{<}\mathtt{P}_1, \ldots, \mathtt{P}_n\text{>}$ and $a \notin \mathtt{S}$: Each $\mathtt{P} \in \{\mathtt{P}_1, \ldots, \mathtt{P}_n\}$ is either a $b$ or a type $\mathtt{G}'$, which does not contain any type placeholders at all. If $b = a$ then we substitute by the Subst-Param rule. After applying Subst-Param consecutively to all parameters of $\mathtt{C}\text{<}\mathtt{P}_1, \ldots, \mathtt{P}_n\text{>}$ we end up with $[\mathtt{G}/a]\mathtt{S} \doteq [\mathtt{G}/a]\mathtt{C}\text{<}\mathtt{P}_1, \ldots, \mathtt{P}_n\text{>}$
3. $\mathtt{N}_1 \doteq \mathtt{N}_2$: Apply Subst-Param to $\mathtt{N}_2$ until we get $\mathtt{N}_1 \doteq [\mathtt{G}/a]\mathtt{N}_2$. Then use Swap rule to get to step 2.
4. $\mathtt{N} \lessdot \mathtt{S}$: It is either a *fail* by the Fail-Subtype rule or we can apply Adapt and continue with step 1 - 3.
5. $a \lessdot \mathtt{S}$: We apply the Subst-Param-Right rule similar to step 3.
6. $\mathtt{S} \lessdot a$: Apply the Subst-Param-Left rule.

□

**Lemma 4.4** (Substitution) *If $C_S$ is a stable model and $(\{a \doteq \mathtt{G}, \overline{a \doteq \mathtt{T}}\} \cup C) \subseteq C_S$ and $C_S$ $\mathtt{ok}$ then $[\overline{\mathtt{T}}/\overline{a}]C \in C_S$.*

*Proof.* By induction over $C' = \overline{a \doteq \mathtt{T}}$.

**Start** $C' = \{a \doteq \mathtt{G}\}$ implies $[\mathtt{G}/a]C$ by lemma 4.3.

**Step** $C' = \{a \doteq \mathtt{G}\} \cup \{\overline{a \doteq \mathtt{T}}\}$. By lemma 4.3 we know that we also have $C'' = \{\overline{a \doteq [\mathtt{G}/a]\mathtt{T}}\}$, because $\{\overline{a \doteq \mathtt{T}}\}$ $\mathtt{ok}$. There must be atleast one $a' \doteq \mathtt{G}' \in C''$, because $C_S$ is a stable model and therefore does not contain a circle (see lemma 4.5). By 4.3 we get a $C''' = [([\mathtt{G}'/a']\overline{\mathtt{T}})/\overline{a}]C$ from $C'' \cup C$.

□

**Lemma 4.5** *A Stable Model does not contain circles: Let $C_S$ be a stable model containing the constraints $\overline{a \doteq \mathtt{T}}$. If $\overline{a}$ are all type placeholders contained in $\overline{\mathtt{T}}$ then the constraints $\overline{a \doteq \mathtt{T}}$ must house atleast one $a \doteq \mathtt{G}$.*

*Proof.* The proof is intermediate from the definition of circles 4.8 and the Fail-Circle rules. There must be either a circle or a $a \doteq \mathtt{G}$ constraint if all type placeholders are involved, otherwise *fail* and $C_S$ is not a stable model. □

The next lemma shows that the Solution-Requirement is enough to ensure a solution set. The algorithm $\Omega$ does not work if a circle is involved. It will *fail* otherwise. So there must exist atleast one $a \doteq \mathtt{G}$ constraint and substituting $\mathtt{G}$ for $a$ leads to atleast one other $b \doteq \mathtt{G}$ constraint and so on. This ensures that a set of $a \doteq \mathtt{N}$ in a stable model also means that $a \doteq \mathtt{G}$ must also be inside this stable model.

**Lemma 4.6** *If $C_S$ is a stable model containing $\{a_1 \doteq \mathtt{N}_1, \ldots, a_n \doteq \mathtt{N}_n\}$ then $\{a_1 \doteq \mathtt{G}_1, \ldots, a_n \doteq \mathtt{G}_n\}$*

*Proof.* Proof by induction.

**Start** $\{a_1 \doteq \mathtt{N}_1\}$ implies $a_1 \doteq \mathtt{G}_1$, because every $b \in \mathtt{N}_1$ leads to $b \doteq \mathtt{Object}$ through S-Object and Solution-Gen, *Note:* $a \notin \mathtt{N}_1$ if $C_S$ is a stable model, due to the Fail-Circle rule.

**Step** $\{a_1 \doteq \mathtt{G}, \overline{a \doteq \mathtt{N}}\}$. By lemma 4.4 we know $[\mathtt{G}/a_1]\overline{a \doteq \mathtt{N}} \in C_S$. By lemma 4.5 there must be atleast one $a \doteq \mathtt{G}$ in $[\mathtt{G}/a_1]\overline{a \doteq \mathtt{N}}$, because $a_1 \cup \overline{a}$ are all type placeholders in $C_S$.

□

**Theorem 1** *Soundness If $C_S$ is a stable model containing $C$ with $C$ ok and containing a set of unifiers $\sigma$ and no fail, then $\forall \mathtt{S} \lessdot \mathtt{T} \in C : \sigma(\mathtt{S}) <: \sigma(\mathtt{T})$ and $\forall \mathtt{S} \doteq \mathtt{T} \in C : \sigma(\mathtt{S}) = \sigma(\mathtt{T})$.*

*Proof.* Additionally we know that for every type placeholder $a$ there is a $\sigma(a) = \mathtt{G} \in C_S$ by lemma 4.1. We prove it for each type of constraint individually:

$\lessdot$ **constraints:** By 4.4 we know that $\overline{\sigma(a)} = \overline{\mathtt{G}}$ and $C$ ok also implies $[\overline{\mathtt{G}}/\overline{a}]C$.

$\doteq$ **constraints** We have $[\overline{\mathtt{G}}/\overline{a}]C$ by 4.4 and we know by lemma 4.1 that $\overline{a}$ are all type placeholders contained in $C$. Therefore the only constraints remaining in $[\overline{\mathtt{G}}/\overline{a}]C$ are the ones of the form $\mathtt{G}_1 \doteq \mathtt{G}_2$ (and $\mathtt{G}_1 \lessdot \mathtt{G}_2$). The Fail-Equals and Reduce rules ensure that $\mathtt{G}_1 \doteq \mathtt{G}_2$ implies either $\mathtt{G}_1 = \mathtt{G}_2$ or *fail*.

□

The following lemma is only true as long as there is no multiple inheritance. This is the case for Featherweight Java, but not for regular Java, where on class can declare multiple interfaces as super types. Lemma 4.7 is rather obvious, because our type system has single inheritance and each class has only one path upward.

**Lemma 4.7** $\mathtt{G} <: \mathtt{G}_1$ *and* $\mathtt{G} <: \mathtt{G}_2$ *means* $\mathtt{G}_1 <: \mathtt{G}_2$ *or* $\mathtt{G}_2 <: \mathtt{G}_1$ *is correct*

*Proof.* Proof by induction over the subtype relations $\mathtt{G} <: \mathtt{G}_1$ (Relation 1) and $\mathtt{G} <: \mathtt{G}_1$ (Relation 2).

Case 1: S-Refl for Relation 1 $\mathtt{G} = \mathtt{G}_1$. Then $\mathtt{G}_1 <: \mathtt{G}_2$ by assumption.

Case 2: S-Refl for Relation 2 $\mathtt{G} = \mathtt{G}_2$. Analogous to Case 1:.

Case 3: S-Class for Relation 1 $\mathtt{G} = \mathtt{C}\langle\overline{\mathtt{G}}\rangle$ in $\mathtt{G} <: \mathtt{G}_1$ and $\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}}\rangle \lhd \mathtt{N}$ with $\mathtt{G}_1 = [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{N}$.

Subcase S-Refl for Relation 2 (see case Case 1:)

Subcase S-Class for Relation 2 means that $\mathtt{G}_1 = \mathtt{G}_2$, because there can only be one class definition for $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$ with one supertype. $\mathtt{G}_1 <: \mathtt{G}_2$ by S-Refl finishes the case.

Subcase S-Trans $\mathtt{G} <: \mathtt{G}'$, $\mathtt{G}' <: \mathtt{G}_2$. Here it must be $\mathtt{G}_1 = \mathtt{G}'$ and therfore $\mathtt{G}_1 <: \mathtt{G}_2$ finishes the case.

This case has three subcases for each variant of Relation 2:

Case 4: S-Trans for Relation 1 $\mathtt{G} <: \mathtt{G}'$, $\mathtt{G}' <: \mathtt{G1}$. This case has three subcases:

Subcase S-Refl for Relation 2 (see case Case 1:)

Subcase S-Class for Relation 2 (see case Case 3:)

Subcase S-Trans for Relation 2 $\mathtt{G} <: \mathtt{G}''$, $\mathtt{G}'' <: \mathtt{G}_2 2$. By i.h. we know from $\mathtt{G} <: \mathtt{G}'$ and $\mathtt{G} <: \mathtt{G}''$ that either $\mathtt{G}' <: \mathtt{G}''$ or $\mathtt{G}'' <: \mathtt{G}'$. Both variants are analogous and we will look at $\mathtt{G}' <: \mathtt{G}''$. Then we have $\mathtt{G}' <: \mathtt{G}_2$ by S-Trans. $\mathtt{G}' <: \mathtt{G}_1$ and $\mathtt{G}' <: \mathtt{G}_2$ lead to either $\mathtt{G}_1 <: \mathtt{G}_2$ or $\mathtt{G}_2 <: \mathtt{G}_1$ finishing the case.

□

**Definition 4.8** *Circular Constraint A set of constraints $\overline{a \doteq \mathtt{N}}$ where $\forall \mathtt{N} \in \overline{\mathtt{N}}$ : fv($\mathtt{N}$) $\neq \emptyset$ and $\overline{a}$ contains all type placeholders used in $\overline{\mathtt{N}}$ ($\overline{a}$ = fv($\overline{\mathtt{N}}$)) is called a circle.*

The algorithm $\Omega$ is not complete in that sense that it does not support F-Bounded types and nested type variables in extends-Relations.

**Theorem 2** *Completeness (partly) The algorithm will find a solution if there exists one (with one exception).*

***Given*** *a $C$ and a $\sigma$,*
*where $C$ ok and $\forall \mathtt{S} \lessdot \mathtt{T} \in C : \sigma(\mathtt{S}) <: \sigma(\mathtt{T})$ and $\forall \mathtt{S} \doteq \mathtt{T} \in C : \sigma(\mathtt{S}) = \sigma(\mathtt{T})$.*

***Then*** *the algorithm will not fail,* ***except*** *when $C$ contains a circle (see definition 4.8)*

*Proof.* We show that none of the implication rules remove a possible solution.

**Solution-Gen** does nothing as long as there is atleast one $a \doteq \texttt{N}$ constraint. The only time this happens is when there is no constraint like $a \doteq \texttt{N}$ or $\texttt{N} \lessdot a$ in the constraint set (Constraints of the form $\texttt{N} \lessdot a$ are turned to $a \doteq \texttt{N}$ by the Super rule). Consequentyl $a$ only appears in a constraint set $C' \subseteq C$ with $C' = \overline{a \lessdot \texttt{N}}$. By assumption we know that there is a $\sigma(a) = \texttt{G}$ with $\forall \texttt{N} \in \overline{\texttt{N}} : \texttt{G} <: \sigma(\texttt{N})$. By lemma 4.7 we know that there must exist one $\sigma(\texttt{N})$ that is a subtype of all other $\overline{\sigma(\texttt{N})}$. Therefore setting $a \doteq \texttt{N}$ will not exclude any solution, as long as we choose the right $\texttt{N}$ out of $\overline{\texttt{N}}$ and our algorithm just tries every possibility, but must pick atleast one induced by the Solution-Requirement rule.

**Solution-Requirement** This rule forces the Solution-Gen rule to create atleast one $a \doteq \texttt{N}$ constraint (see Solution-Gen case).

**Super** We assume that all `extends`-Relations are well-formed (see rule WF-Class), which means that every supertype of a well-formed type $\texttt{N}$ is well-formed aswell. Super checks all possible supertypes.

**Reduce** If $\sigma(\texttt{C<}\texttt{P}_1, \ldots \texttt{P}_n\texttt{>}) = \sigma(\texttt{C<}\texttt{P'}_1, \ldots \texttt{P'}_n\texttt{>})$ then also $\sigma(\texttt{P}_1) = \sigma(\texttt{P'}_1), \ldots, \sigma(\texttt{P}_n) = \sigma(\texttt{P'}_n)$.

**Adapt** Same as Reduce case.

**Subst-Param** and all other Subst-Cases. If $a \doteq \texttt{G} \in C$ then $\sigma(a) = \texttt{G}$, because $\sigma$ is a valid solution. Knowing $\sigma(a) = \texttt{G}$ we can replace all occurences of $a$ by $\texttt{G}$ without excluding any solution.

**S-Object** All classes in Java are a subtype of `Object`. $\sigma(a) <: \texttt{Object}$ holds by definition of Java subtyping.

**Swap** Intermediate.

**Solution** Intermediate.

□

## 5. ASP Encoding

**See appendix A for the complete program including an example input**

ASP statements consist of a head and a body separated by a implication operator ":-": `head :- body`. This statement is interpreted as an implication rule. If the premises in the body are satisfied the facts stated in the head are deducted. The implication rules defined in figure 4 are formulated in a way that can be translated to an ASP program. This chapter explains how to decode our algorithm $\Omega$ in ASP. At first we give a ASP representation for each syntax element used in figure 3:

| Syntax element | ASP representation |
|---|---|
| $\ldots \lessdot \ldots$ | `subcons(..., ...)` |
| $\ldots \doteq \ldots$ | `equalcons(..., ...)` |
| $a$ | `tph("a")` |
| `C<...>` | `type("C", params(...))` |
| `Object` | `type("Object", null)` |

For further clarification we will present some examples:

$$\texttt{C<Object>} \implies \texttt{type"C", params(Object,null)}$$
$$a \lessdot b \implies \texttt{subcons(tph("a"), tph("b"))}$$
$$\texttt{List<}a\texttt{>} \implies \texttt{type("List", params(tph("a")))}$$

The subtype rules in figure 4 require a special treatment, because the S-Class rule contains a substitution which has to be encoded using variables. Given a extends relation `class C<X>` $\lhd$ `D<X>` we generate the ASP code:

```
subtype(type("C", params(X)),
type("D", params(X)))
    :- subtype(type("C", params(X))).
```

Capital letters like `X` are variables in ASP and the former statement causes any literal like `subtype(type("C", params(tph("a"))))` to imply the literal `subtype(type("C", params(tph("a"))), type("D", params(tph("a")))).`

The vital part is the generation of the cartesian product of the Or-Constraints.
`subcons(A,B); orCons(C,D) :- orCons(subcons(A,B), subcons(C,D)).`
The operator ";" tells ASP to consider either the left or the right side. By supplying multiple Or-Constraints this way the ASP interpreter will consider all combinations, the cartesian product of the or constraints.

The rules implying a failure $\emptyset$ are translated by leaving the head of the asp statement empty. If the body of such rules is satisfied the algorithm considers the deduction as incorrect. See an example in listing 1 that shows an ASP encodign of the Fail-Equals rule.

**Listing 1:** Fail-Equals rule in ASP

```
:- equalcons(type(A, _),
    type(B, _)),A != B.
```

## 6. Outcome and Conclusion

ASP handles Or-Constraints surprisingly well. We tested our ASP implementation of the unification algorithm with the constraints originating from the program in figure 2. We compared it with the current implementation of the Unification algorithm in Java.[1] We increased the complexity by adding more stacked method calls up to ten stacked calls and the interpreter clingo[2] was still able to handle it easily and finish computation in under 50 milliseconds. By contrast our Java implementation already takes multiple seconds processing time for the same input.

We are using this example for a fair comparision between the two implementations because it does not include generic type parameters causing the unification algorithm not to consider wildcard types. The Java implementation also supports Java wildcard types whereas the ASP implementation does not. It is unfortunately not possible to implement the unification algorithm including wildcard support with Answer Set Programming. The reason is that subtype checking in Java is turing complete [1]. While the grounding process of the ASP interpreter clingo would not terminate if the problem is turing complete [2]; the Java implementation is still able to spawn atleast one solution in most cases and only never terminates for specific inputs, whereas the ASP program never terminates as soon as wildcards are involved.

## References


[1] Radu Grigore. “Java generics are turing complete”. In: *ACM SIGPLAN Notices* 52.1 (2017), pp. 73–85.

[2] Benjamin Kaufmann et al. “Grounding and solving in answer set programming”. In: *AI magazine* 37.3 (2016), pp. 25–32.

[3] Martin Plümicke. “Java Type Unification with Wildcards”. In: *Applications of Declarative Programming and Knowledge Management, 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*. Ed. by Dietmar Seipel, Michael Hanus, and Armin Wolf. Vol. 5437. Lecture Notes in Computer Science. Springer, 2007, pp. 223–240.

[4] Martin Plümicke. “Optimization of the Java Type Unification”. In: *Proceedings of the 37th Workshop on (Constraint) Logic Programming (WLP 2023)*. Ed. by Sibylle Schwarz and Mario Wenzel. 2023. URL: https://dbs.informatik.uni-halle.de/wlp2023/WLP2023_Pl%C3%BCmicke_Optimization%20of%20the%20Java%20Type%20Unification.pdf.

[5] Martin Plümicke. “Optimization of the Java type unification”. In: *Proceedings of the 35th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts*. Ed. by Jens Knoop, Martin Steffen, and Baltasar Trancón y Widemann. Research Report 482. ISBN 978-82-7368-447-9. Faculty of Mathematics and Natural Sciences, UNIVERSITY OF OSLO. 2018, pp. 3–12.

[6] Andreas Stadelmeier, Martin Plümicke, and Peter Thiemann. “Global Type Inference for Featherweight Generic Java”. In: *CoRR* abs/2205.08768 (2022). arXiv: 2205.08768. URL: https://doi.org/10.48550/arXiv.2205.08768.


[1] https://gitea.hb.dhbw-stuttgart.de/JavaTX/JavaCompilerCore

[2] https://potassco.org/clingo/

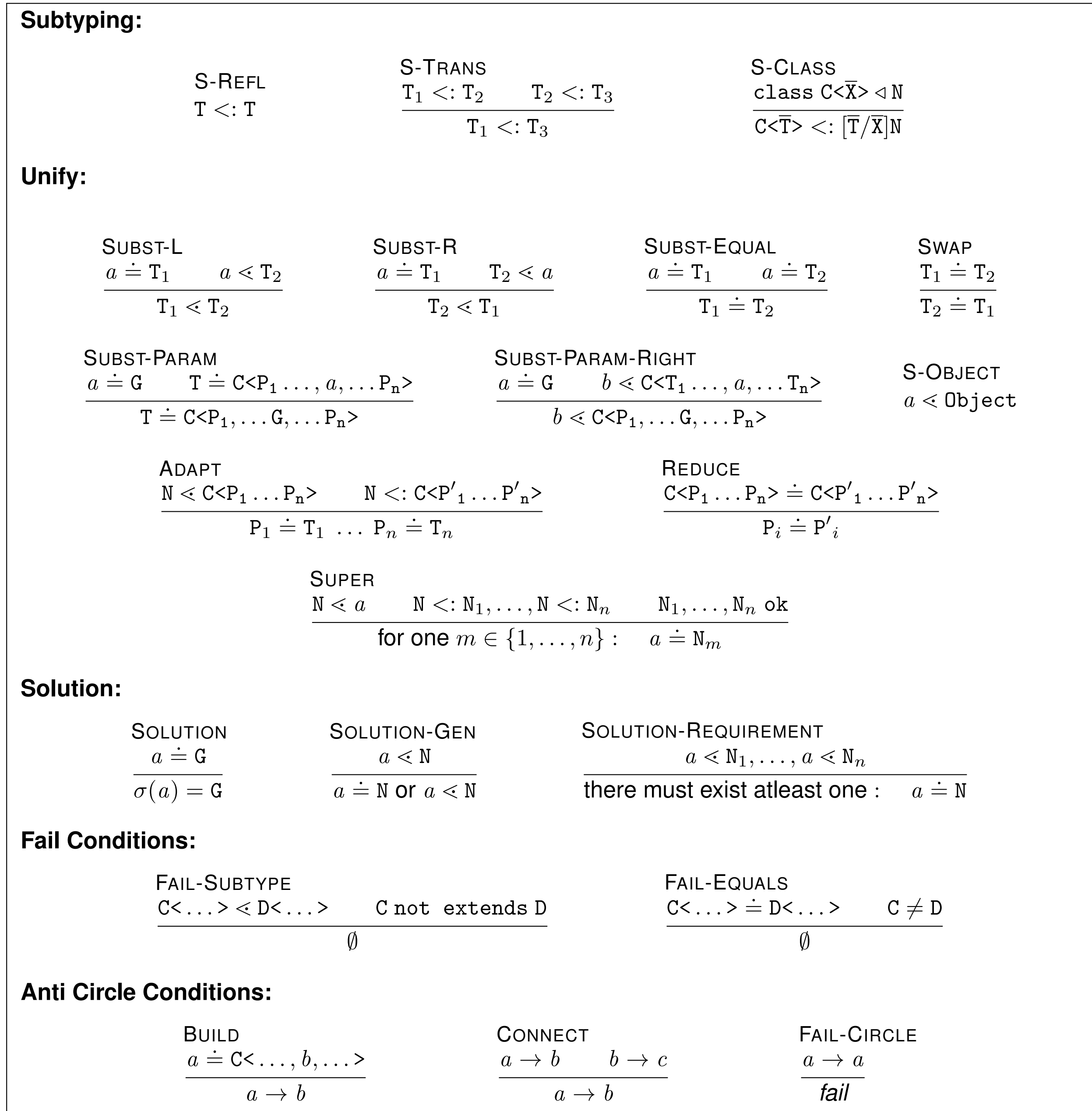


**Figure 4:** Unify Algorithm $\Omega$

# A. ASP Program + Example Input

```
% TEST INPUT

subtype(type("java.lang.Boolean",null),type("java.lang.Object",null)):-subtype(type("java.lang.
    Boolean",null)).subtype(type("java.lang.Integer",null),type("java.lang.Number",null)):-
    subtype(type("java.lang.Integer",null)).subtype(type("java.lang.Number",null),type("java.
    lang.Object",null)):-subtype(type("java.lang.Number",null)).
subtype(type("java.util.Vector",params(XX)),type("java.lang.Object",null)):-subtype(type("java.
    util.Vector",params(XX))).
subtype(type("Matrix",null),type("java.util.Vector",params(type("java.util.Vector",params(type(
    "java.lang.Integer",null)))))):-subtype(type("Matrix",null)).

subtype(type("java.lang.Boolean",null),type("java.lang.Object",null)):-subtype(type("java.lang.
    Boolean",null)).subtype(type("java.lang.Integer",null),type("java.lang.Number",null)):-
    subtype(type("java.lang.Integer",null)).subtype(type("java.lang.Number",null),type("java.
    lang.Object",null)):-subtype(type("java.lang.Number",null)).subtype(type("java.util.Vector
    ",params(XX)),type("java.lang.Object",null)):-subtype(type("java.util.Vector",params(XX)))
    .subtype(type("Matrix",null),type("java.util.Vector",params(type("java.util.Vector",params
    (type("java.lang.Integer",null)))))):-subtype(type("Matrix",null)).

subtype(type("C1",null),type("java.lang.Object",null)):-subtype(type("C1",null)).
subtype(type("C2",null),type("java.lang.Object",null)):-subtype(type("C2",null)).
subtype(type("java.lang.Boolean",null),type("java.lang.Object",null)):-subtype(type("java.lang.
    Boolean",null)).subtype(type("java.lang.String",null),type("java.lang.Object",null)):-
    subtype(type("java.lang.String",null)).subtype(type("java.lang.Integer",null),type("java.
    lang.Object",null)):-subtype(type("java.lang.Integer",null)).subtype(type("OrConsTest",
    null),type("java.lang.Object",null)):-subtype(type("OrConsTest",null)).subtype(type("
    MyPair",params(XX,XY)),type("Pair",params(XX,XX))):-subtype(type("MyPair",params(XX,XY))).
    subtype(type("Pair",params(XX,XY)),type("java.lang.Object",null)):-subtype(type("Pair",
    params(XX,XY))).subtype(type("List",params(XX)),type("java.lang.Object",null)):-subtype(
    type("List",params(XX))).subtype(type("Integer",null),type("java.lang.Object",null)):-
    subtype(type("Integer",null)).subtype(type("String",null),type("java.lang.Object",null)):-
    subtype(type("String",null)).


% Or-Constraints
undCons(A,B) :- orCons(undCons(A,B), null).
undCons(A,B); orCons(C,D) :- orCons(undCons(A,B), orCons(C,D)).

% undCons
subcons(A,B) :- undCons(subcons(A,B), _).
undCons(C,D) :- undCons(_, undCons(C,D)).
equalcons(A,B) :- undCons(equalcons(A,B), _).
undCons(B,C) :- undCons(A, undCons(B, C)).

% Subtyping
subtype(A, A) :- subtype(A). % reflexive
subtype(A, A) :- subtype(A, B). % reflexive
subtype(B) :- subtype(A, B). % transitive
subtype(A, C) :- subtype(A, B), subtype(B, C). % transitive
% named Subtyping
namedSubtype(A,B) :- subtype(type(A, AP), type(B, BP)).
% is reflexive and transitive because subtype it stems from subtype

% generate the subtype relations for every constraint where one is needed: (this could be
    optimized)
subtype(type(A, P)) :- subcons(_, type(A, P)).
subtype(type(A, P)) :- subcons(type(A, P), _).
```

```
% subst:
subst(tph(A), type(N,P)) :- sigma(tph(A), type(N, P)).

% subst-L:
subcons(B,C) :- subst(A, B), subcons(A, C).
%subst-R:
subcons(C,B) :- subst(A, B), subcons(C, A).
%subst-Equal:
equalcons(B,C) :- subst(A, B), equalcons(A, C).

%swap:
equalcons(A,B) :- equalcons(B,A).


%S-Object
subcons(tph(A), type("java.lang.Object", null)) :- subcons(tph(A), _).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(_, tph(A)).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(_, type(_, params(tph(A)))).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(_, type(_, params(tph(A), _))).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(_, type(_, params(_, tph(A)))).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(_, type(_, params(tph(A), _, _))).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(_, type(_, params(_, tph(A), _))).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(_, type(_, params(_, _, tph(A)))).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(type(_, params(tph(A))), _).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(type(_, params(tph(A), _)), _).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(type(_, params(_, tph(A))), _).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(type(_, params(tph(A), _, _)), _).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(type(_, params(tph(A), _, _)), _).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(type(_, params(_, tph(A), _)), _).
subcons(tph(A), type("java.lang.Object", null)) :- subcons(type(_, params(_, _, tph(A))), _).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(tph(A), _).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(_, tph(A)).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(_, type(_, params(tph(A)))).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(_, type(_, params(tph(A), _))).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(_, type(_, params(_, tph(A)))).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(_, type(_, params(tph(A), _, _))).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(_, type(_, params(_, tph(A), _))).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(_, type(_, params(_, _, tph(A)))).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(type(_, params(tph(A))), _).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(type(_, params(tph(A), _)), _).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(type(_, params(_, tph(A))), _).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(type(_, params(tph(A), _, _)), _).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(type(_, params(tph(A), _, _)), _).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(type(_, params(_, tph(A), _)), _).
subcons(tph(A), type("java.lang.Object", null)) :- equalcons(type(_, params(_, _, tph(A))), _).

% Subst-Param:
equalcons(A, type(C, params(T2))) :- equalcons(A, type(C, params(T))), subst(T, T2).
equalcons(A, type(C, params(T2, P2))) :- equalcons(A, type(C, params(T, P2))), subst(T, T2).
equalcons(A, type(C, params(P1, T2))) :- equalcons(A, type(C, params(P1, T))), subst(T, T2).
equalcons(A, type(C, params(T2, P2, P3))) :- equalcons(A, type(C, params(T, P2, P3))), subst(T,
     T2).
equalcons(A, type(C, params(P1, T2, P3))) :- equalcons(A, type(C, params(P1, T, P3))), subst(T,
     T2).
equalcons(A, type(C, params( P1, P2, T2))) :- equalcons(A, type(C, params(P1, P2, T))), subst(T
    , T2).

% Subst-Param-Right:
subcons(A, type(C, params(T2))) :- subcons(A, type(C, params(T))), subst(T, T2).
subcons(A, type(C, params(T2, P2))) :- subcons(A, type(C, params(T, P2))), subst(T, T2).
```

```
subcons(A, type(C, params(P1, T2))) :- subcons(A, type(C, params(P1, T))), subst(T, T2).
subcons(A, type(C, params(T2, P2, P3))) :- subcons(A, type(C, params(T, P2, P3))), subst(T, T2)
    .
subcons(A, type(C, params(P1, T2, P3))) :- subcons(A, type(C, params(P1, T, P3))), subst(T, T2)
    .
subcons(A, type(C, params( P1, P2, T2))) :- subcons(A, type(C, params(P1, P2, T))), subst(T, T2
    ).

% reduce
equalcons(P1, PP1) :- equalcons(type(C, params(P1)), type(C, params(PP1))).
equalcons(P1, PP1) :- equalcons(type(C, params(P1, P2)), type(C, params(PP1, PP2))).
equalcons(P2, PP2) :- equalcons(type(C, params(P1, P2)), type(C, params(PP1, PP2))).
equalcons(P1, PP1) :- equalcons(type(C, params(P1, P2, P3)), type(C, params(PP1, PP2, PP3))).
equalcons(P2, PP2) :- equalcons(type(C, params(P1, P2, P3)), type(C, params(PP1, PP2, PP3))).
equalcons(P3, PP3) :- equalcons(type(C, params(P1, P2, P3)), type(C, params(PP1, PP2, PP3))).

% super
{ equalcons(tph(A), type(D, DP)):  subtype(type(C, CP), type(D, DP)) } == 1 :- subcons(type(C,
    CP), tph(A)).

% Solution-Gen
{ equalcons(tph(A), type(T,P)) } :-  subcons(tph(A), type(T,P)).

% Solution-Requirement
:- subcons(tph(A), _), not sigma(tph(A), _).

%Solution:
tphs( P ) :- equalcons(tph(A), type(C, P)).
sigma(tph(A), type(C,P)) :- equalcons(tph(A), type(C, P)), not tphs(_, P).

% fail-equals
:- equalcons(type(C, CP), type(D, DP)), C != D.

% fail-subtype
:- subcons(type(C, CP), type(D, DP)),  not namedSubtype(C, D).

% Build
connected(A,B) :- equalcons(tph(A), type(_,params(tph(B)))).
connected(A,B) :- equalcons(tph(A), type(_,params(tph(B),_))).
connected(A,B) :- equalcons(tph(A), type(_,params(_,tph(B)))).
connected(A,B) :- equalcons(tph(A), type(_,params(tph(B),_,_))).
connected(A,B) :- equalcons(tph(A), type(_,params(_,tph(B),_))).
connected(A,B) :- equalcons(tph(A), type(_,params(_,_,tph(B)))).
%connect
connected(A,C) :- connected(A,B), connected(A,C).
%Fail-circle
:- connected(A,A).


%% Helpers
%tphs:
tphs(tph(P), params(tph(P))) :- tphs( params(tph(P))).
tphs(P) :- tphs( params(type(C, P))).
tphs(tph(A), params(type(C, P))) :- tphs( params(type(C, P))), tphs(tph(A), P).

tphs(tph(P), params(X, tph(P))) :- tphs( params(X, tph(P))).
tphs(tph(P), params(tph(P), X)) :- tphs( params(tph(P), X)).
tphs(P) :- tphs( params(X, type(C, P))).
tphs(P) :- tphs( params(type(C, P), X)).
tphs(tph(A), params(X, type(C, P))) :- tphs( params(X, type(C, P))), tphs(tph(A), P).
```

```
tphs(tph(A), params(type(C, P), X)) :- tphs( params(type(C, P), X)), tphs(tph(A), P).


tphs(tph(P), params(tph(P), X, Y)) :- tphs( params(tph(P), X, Y)).
tphs(tph(P), params(X, tph(P), Y)) :- tphs( params(X, tph(P), Y)).
tphs(tph(P), params(X,Y,tph(P))) :- tphs( params(X, Y, tph(P))).
tphs(P) :- tphs( params(type(C, P), X, Y)).
tphs(P) :- tphs( params(X, type(C, P),Y)).
tphs(P) :- tphs( params(X,Y,type(C, P))).
tphs(tph(A), params(type(C, P), X, Y)) :- tphs( params(type(C, P), X, Y)), tphs(tph(A), P).
tphs(tph(A), params(X, type(C, P), Y)) :- tphs( params(X, type(C, P), Y)), tphs(tph(A), P).
tphs(tph(A), params(X, Y, type(C, P))) :- tphs( params(X, Y, type(C, P))), tphs(tph(A), P).


#show sigma/2.
%#show sigma2/2.
```

# The Java-TX Roadmap

Martin Plümicke
Duale Hochschule Baden-Württemberg (DHBW) Stuttgart Campus Horb
Department of Computer Science
Florianstraße 15, 72160 Horb
pl@dhbw.de

## Abstract

Java-TX is a language based on Java. The predominant new features are global type inference and real function types for lambda expressions. In recent years theory and implementation was done, such that a compiler is available.
There are some further projects to refine Java-TX. We give an overview in this paper:

**Pattern Matching:** In the latest versions of Java record types and pattern matching has been introduced. In combination with global type inference pattern matching can be extended such that pattern matching is similar to Haskell (especially pattern matching in method headers).

**Generated Generics:** As result of the type inference algorithm constraints of two type variables ($a \lessdot b$) could arise. Generally, these constraints are transferred to bound type variables. Furthermore, type variables in Java-TX are only reflexive if they are declared as reflexive `<X extends X>`. Declarations like this are not allowed in Java. Therefore, a Java-TX-Signature has to be introduced in Java-TX.

**Real Functiontypes in Java-TX:** The implementation of the strawman approach (integration of real function types and functional interfaces) has to be completed.

**Heterogeneous Translation:** In Java type erasure erases all parameters of types during compilation. From this follows that some results of the type inference could not be translated into byte-code. Therefore, heterogeneous translation is needed.

**Type Inference of Bound Generics:** At the moment Java-TX allows bound type variables (`<A extends` $ty$`>`) only if the bound is a type variable, too. The introduction of other bounds would reduce the runtime of the type inference algorithm.

**Unifications algorithm as a Webservice:** We wolud like to offer the unification algorithmus as webservice on a powerful parallel system, such that complex type constraints can be solved even if the client is not powerful.

**Java-TX-compiler in Java-TX:** A first large Java-TX-project is the implementation of a Java-TX-compiler.

**Module System for Java-TX:** A new challenge is to transfer ideas from the complex SML module system to Java-TX.

## 1 Introduction

Java-TX (i.e. **T**ype e**X**tended) [6] is a language based on Java. The predominant new features are global type inference and real function types for lambda expressions. Global type inference means that all type annotations can be omitted, and the compiler infers them without losing the static type property.

Function types are introduced in a similar fashion as in Scala but additionally integrated them into the Java target-typing as proposed in the so-called strawman approach.

The language Java-TX corresponds to Java in version 8. Apart from some trivia, we reduced the language currently by two essential features, exceptions and generics bound by non type-variable types (only type variables as bounds are allowed). Furthermore, basic types (`int`, `float`, `bool`, `...`) were left out, such that one has to use the boxed variants. Literals as `1`, `2`, `3`, `...`, `true`, `false` are still allowed. All type annotations are optional. They can be inferred by our type inference algorithm.

The Java-TX type system corresponds closely to the original Java 8 type system described in [1]. However, we have extended it with real function types [5].

In Java 14 record types and switch expressions were introduced. The record types are the basis of pattern matching in Java.

Finally, pattern matching has been introduced in the versions 16 - 22. In contrast to functional programming languages, all pattern-variables must have a type annotation and it is not allowed to give patterns in method headers. Both caveats are addressed in Java-TX.

We have developed a prototypical implementation and a language server plugin for common IDEs [3].

In this paper, we present an overview of the Java-TX roadmap.

## 2 Pattern matching

In the latest versions of Java record types and pattern matching have been introduced. In combination with global type inference, pattern matching can be extended to provide Haskell-like functionality (especially pattern matching in method headers).

For clarification, consider the example in Listing 1. First, a sealed interface `List` is declared. Sealed

```
sealed interface List<T>
permits Cons, Empty {}

public record Cons<T>(T a, List<T> l)
implements List<T> {}

public record Empty<T>()
implements List<T> {}

public class PatternMatchingListAppend {

  public append(Cons(a, b), list2) {
      return
        new Cons<>(a,append(b,list2));
  }

  public append(Empty(), list2) {
    return list2;
  }
}
```

**Listing 1:** Pattern Matching in Java-TX

interfaces restrict which types are permitted to implement the interface, in the `permits`-clause. In this example `Cons` and `Empty` are the only types which implement `List`. Sealed interfaces in Java correspond to the data- and newtype-declaration, respectively, in Haskell.

The Java-TX-class `PatternMatchingListAppend` contains the well-known `append` method in a Haskell-like style. Note that pattern matching is used in method headers and no type declarations are given. The goal is to compile code as given in Listing 2. In Haskell both append-declarations

```
<T> Cons append(Cons<T> l1, List<T> l2) {
  return switch(l1) {
    case Cons(T a, Cons<T> b)
      -> new Cons(a, append(b, l2));
    case Cons(T a, Empty<T> b)
      -> new Cons(a, ❶append(b, l2));
  }
}
<T> Cons append(Empty<T> l1, List<T> l2) {
  return l2;
}
```

**Listing 2:** Compiled code of `append`

would be a single case-by-case defined function

```
class TPHsAndGenerics {

  id = x -> x;

  id2(x){
      return id.apply(x);}
  m(a, b){
      return b; }

  m2(a, b) {
      var c = m(a,b);
      return a; }
}
```

```
class TPHsAndGenerics {
  Fun1$$<UD, ETX>
      id = (DZP x) -> x;

  ETX id2(V x) {
       return id.apply(x);}
  AI m(AM a, AN b){
      return b; }

  AA m2(AB a, AD b){
      AE c = m(a,b);
      return a; }
}
```

$$\{\,\mathtt{AB} \lessdot \mathtt{AA}, \mathtt{AD} \lessdot \mathtt{AN}, \mathtt{AN} \lessdot \mathtt{AI}, \mathtt{AI} \lessdot \mathtt{AE}, \mathtt{V} \lessdot \mathtt{UD}, \mathtt{AB} \lessdot \mathtt{AM}, \mathtt{DZP} \lessdot \mathtt{ETX}, \mathtt{UD} \lessdot \mathtt{DZP}\,\}$$

**Figure 1:** Java-TX source code and the result of type inference algorithm

with domain `List`×`List`. In contrast in Java-TX, it is represented as two (overloaded) functions with the domain `Cons`×`List` and `Empty`×`List`, respectively. Therefore the function-call ❶ could call both functions depending on the type of the argument. Calling the method with the domain `Empty`×`List` represents the base case and therefore terminates the recursion.

Additionally, in Java-TX a new bridge-method with the domain `List`×`List` should be generated which calls the corresponding method.

A more detailed consideration of pattern matching in Java-TX is given in these proceedings in [2].

## 3 Generated Generics

One of the main features of Java-TX is global type inference. The type inference algorithm [8] consists of two steps, the tree traversal and the type unification [7]. During traversal of the abstract syntax tree, type constraints are generated. These type constraints are then resolved by the type unification. The result of the type unification is a finite number of mostly general unifiers and a set of remaining constraints, consisting only of two type variables ($a \lessdot b$). Generally, these constraints are transferred to bound type variables `<a extends b>`. In some cases this turns out to be impossible (cycles, infima). These constraints have to be reduced in such a way, that the principal type is not restricted.

The algorithm consists of four steps:

- Building the family of generated generics of the class and its methods
- Complete the family of generated generics
- Eliminating the cycles
- Eliminating the infima

Let us consider an example for the first two steps of the generics algorithm. In Figure 1 the Java-TX source program and the result of the type inference algorithm are given. In Listing 3 the completed family of generated generics is given.

```
class TPHsAndGenerics
  <UD extends DZP, DZP extends ETX, ETX> {

  Fun1$$<UD, ETX> id = x -> x;

  <V extends UD> ETX id2(V x) {
      return id.apply(x);}

  <AM, AN extends AI, AI>
  AI m(AM a, AN b){
      return b;}

  <AA,AB extends AA, AD extends AE, AE>
  AA m2(AB a, AD b){
      AE c = m(a,b);
      return a;}
}
```

**Listing 3:** Completed family of generated generics

The generics are divided in class generics (`UD`, `DZP`, and `ETX` and generics of the methods (`V` of `id2`, `AM`, `AN`, and `AI` of `m`, and `AA`, `AB`, `AD`, and `AE` of `m2`). Bounds of the method generics can be either a class generic (`UD` is a bound of `V`) or other generics of the same method. In this example the latter is the case.

The next examples show how Java-TX eliminates cycles in generics. Let the class `Box` in Listing 4 with type annotations, be given.

```
class Box<T> {
  T elem;

  T get() { return elem; }

  void set(T x) { elem = x; }
}
```

**Listing 4:** The class `Box`

Furthermore an untyped method `m` is given as

```
  m(b1, b2) {
    b1.set(b2.get());
    b2.set(b1.get());
  }
```

Type inference and the first two steps of the algorithm results are shown in Listing 5.

```
  <L extends M, M extends L>}
  void m(Box <L> b1, Box<M> b2) {
    b1.set(b2.get());
    b2.set(b1.get());
  }
```

**Listing 5:** Cycle in generics

The problem is, that `<L extends M, M extends L>` is not correct. In this case, we introduce a fresh type variable which replaces all type variables of the cycle (Listing 6).

For more details (especially eliminating infima) we refer to [10].

## 3.1 Fully-fledged Wildcards in Java-TX

In Java-TX the use of wildcards is changed. Wildcards are full-fledged types and can be used at any position as all other types. The subtyping relation is extended by the following semantics idea:

$?\,\texttt{extends}\,\theta$**:** *There is a subtype of* $\theta$*.*

$?\,\texttt{super}\,\theta'$**:** *There is a supertype of* $\theta'$*.*

```
  <L> void m(Box <L> b1, Box<L> b2) {
    b1.set(b2.get());
    b2.set(b1.get());
  }
```

**Listing 6:** Introduction of a fresh type variable `L` to eliminate cycles in generics

**Definition 3.1** (Subtyping Relation on wildcards)

*For two Java-TX-types* $\theta <: \theta'$ *holds*

$$\theta \;<:\; ?\,\texttt{super}\,\theta' \text{ and}$$
$$?\,\texttt{extends}\,\theta \;<:\; \theta'.$$

From this follows $?\,\texttt{extends}\,\theta \;<:\; ?\,\texttt{super}\,\theta'$ and especially: `<:` is not reflexive on wildcards.

Let us consider Listing 4 again, under the assumption that Listing 5 is correct. In the following snippet

```
   Box<?> bb1 = ...;
   Box<?> bb2 = ...;
   m(bb1, bb2);
...
```

the call of `m` would not be correct as the instantiation of `?` for `L` and `M`, respectively, would lead to `? <: ?`, which is not correct.

Considering Listing 6 this would also mean that `<:` could not be reflexive on type variables. Therefore, type variables in Java-TX are only reflexive if they are declared reflexive, i.e. `<X extends X>`.

Declarations like this are not allowed in Java. Especially, a cyclic declaration of generic bounds leads to the crash of the JVM.

Therefore in Java-TX, additional Java-TX-signatures have to be introduced

`Java-TX-signature: <L extends L>`

which are type checked during the method call is type checked.

Consider the following slightly varied example of our Box class

```
class Box<T> {
  T elem;

  T get() { return elem; }

  void set(T x) { elem = x; }
}

  void m(Box <T> b) {
```

```
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.control.Button;
...
Button btn = new Button();
btn.setText("Say␣'Hello␣World'");
btn.setOnAction(
  event -> System.out.println("Hello␣World!")
);
```

```
Fun1$$<ActionEvent, String> helloworld
 = event -> System.out.println("Hello␣World!");
Button btn = new Button();
btn.setText("Say␣'Hello␣World'");
btn.setOnAction(helloworld);
```

**Figure 2:** Function types in Java and Java-TX

```
    b.set(b.get());
  }
...
   Box<?> bb = ...;
   m(bb);
...
```

Now the call of `m` is sound. In this case no additional

`Java-TX-signature`: `<L extends L>`

is not necessary!

Summarized, the idea of implementing generics in the Java-TX type system with full-fledged wildcards is:

- During traversal of the abstract syntax in the type inference algorithm, any node gets either the known/declared type or pairwise different type placeholders.
- If during cycle or infima eliminating type placeholders are coincided in generics declaration, an additional reflexive bound is added as Java-TX-signature.
- Type variables in annotated types can only be used once.

## 4 Real function types in Java-TX

### 4.1 Special Java-TX functional interfaces

In [5] we gave the theory of Java-TX function types. The function types are special functional interfaces like function types Scala. Additionally, we realized the so called strawman approach [11]. In comparison to the Java approach of lambda expression with functional interface as target types of lambda expressions the Java-TX approach types lambda expressions by function types and allows subtyping of function types and direct application of lambda expressions without type-casts. The implementation of the strawman approach (integration of real function types and functional interfaces) allows the implementation of SAM-types (essentially interfaces with one method) by all expressions with a compatible function type.

In Figure 2 the implementation of a JavaFXexample, borrowed from Oracle's getting-started-tutorial, illustrates the situation: On the left hand the Java implementation is given. The lambda expression

```
event -> System.out.println("Hello␣World!")
```

as argument of the method `setOnAction` implements the functional interface `EventHandler` where the untyped lambda expression is (type) compatible to the method `handle` in the functional interface `EventHandler`

On the right hand side the Java-TX implementation is given. The typed lambda expression is assigned to the local variable `helloworld`. After that the method `setOnAction` is called with the argument `helloworld`. In Java this would not be possible, as the interface `Fun1$$` is not compatible to the interface `EventHandler`.

### 4.2 Heterogeneous translation of functional interfaces

In Java type erasure erases all parameters of types during compilation. From this follows that some results of the type inference could not be translated into byte-code.

Therefore, we have introduced heterogeneous translation for the special Java-TX functional interfaces [14].

The following example shows the feature. Suppose the operator + is overloaded by `Double`,

```
class OLFun {
  m(f) {
    var x;
    x = f.apply(x+x);
    return x;
  }
}
```

Integer, and String. Therefore, the function f would be overloaded by

$$\texttt{Double} \to \texttt{A} \\ \& \texttt{Integer} \to \texttt{A} \\ \& \texttt{String} \to \texttt{A}$$

But the descriptors generated a Java-TX compiler with homogeneous translation would be ambiguous:

```
<A> A m(Fun1$$ <Double, A>);
     descriptor: (LFun1$$;)Object;
<A> A m(Fun1$$ <Integer, A>);
     descriptor: (LFun1$$;)Object;
<A> A m(Fun1$$ <String, A>);
     descriptor: (LFun1$$;)Object;
```

The parameters of the functional interface would be erased.

In contrast in Java-TX $-separated arguments to the special functional interfaces are added:

```
<A> A m(Fun1$$ <Double,A>);
 descriptor: (LFun1$$$_$Double$_$LTPH$_$;)Object;
<A> A m(Fun1$$ <Integer,A>);
 descriptor: (LFun1$$$_$Integer$_$LTPH$_$;)Object;
<A> A m(Fun1$$ <String,A>);
 descriptor: (LFun1$$$_$String$_$LTPH$_$;)Object;
```

Therefore the class OLFun becomes a (type-) correct Java-TX-class.

In the future the implementation of the Java-TX function types has to be completed, where we will orient our implementation to the implementation of function types in *Kotlin*.

## 5 Heterogeneous translation

In subsection 4.2 we described the heterogeneous translation of special functional interfaces. There are similar effects in further generic types. Let aus consider the class VectorAdd. The JVM descriptors generated by a Java-TX-compiler without heterogeneous translation of function types, erases the type parameters of the types of v1 and v2. Therefore the method m is ambiguous.

```
class VectorAdd {
  m(v1, v2) {
    var ret = new Vector<>();
    while (i < v1.size()) {
      erg.addElement(v1.elementAt(i)
                   + v2.elementAt(i));
      i++;
    }
  }}
```

```
Double m(Vector<Double>, Vector<Double>);
    descriptor:(LVector;LVector;)LVector;
Integer m(Vector<Integer>, Vector<Integer>);
    descriptor:(LVector;LVector;)LVector;
String m(Vector<String>, Vector<String>);
    descriptor:(LVector;LVector;)LVector;
```

In the future, a similar approach as in the special functional interfaces should be transferred to all generic types in Java-TX. However, there are some challenges to this. Nonetheless, there is a first approach to this general heterogeneous compilation introduced by PIZZA [4].

## 6 Type Inference of Bound Generics

At the moment Java-TX allows bound type variables (<A extends *ty*>) only if the bound is a type variable, too. The introduction of other bounds would reduce the run-time of the type inference algorithm.

Let us consider the class Pair. The type inference and the process of generated generics lead to the program in Listing 7. If we add the method m

```
m() { Number n = getfst(); }
```

```
class Pair<X> {
  X fst;
  X snd;

  Pair(X fst, X snd) { this.fst=fst;
                       this.snd=snd; }

  X getfst() { return this.fst; }

  Pair<X> swap() {
    return new Pair<>(this.snd, this.fst);
  }
}
```

**Listing 7:** The class Pair

to the typeless version of `Pair`, the constraint $\{ \texttt{X} \lessdot \texttt{Number} \}$ is generated. This induces the result of the type inference $\texttt{X} = \texttt{Number}$.

This means that the class `Pair` is no longer generic and instead the type variable `X` is given the type `Number`.

In [13], we extended the type inference algorithm such that constraints like $\{ \texttt{X} \lessdot \texttt{Number} \}$ will be unchanged. This would lead to a bound generic `<X extends Number>` as given in Listing 8.

```
class Pair<X extends Number> {
  X fst;
  X snd;

  Pair(X fst, X snd) { this.fst=fst;
                       this.snd=snd; }

  X getfst() { return this.fst; }

  Pair<X> swap() {
    return new Pair<>(this.snd, this.fst);
  }
  void m() { Number n = getfst(); }
}
```

**Listing 8:** Generic Pair with bound parameter

In future versions of the Java-TX-compiler, we intend to implement this new feature.

## 7 Java-TX–Compiler in Java-TX

In a student research project a part of the existing Java-TX-Compiler — implemented in Java — was translated to Java-TX [12]. As Java-TX compiles to bytecode just as original Java, the codebase can be translated class by class, preserving the usability of the whole compiler and allowing for iterative testing.

The following steps have to be done:

**Completion of the implementation of the Java-TX–compiler in Java-TX:** The challenge of this step is, that the actual Java-TX-compiler is indeed turing-complete, but not all Java features are implemented yet. This means, either the original Java code must be rewritten such that only implemented Java-TX features are used, or the Java-TX–compiler has to be extended. We try using the second possibility, such that the Java-TX-compiler becomes more and more complete.

**Remove type annotations in the Java code:** Basically, it would be possible to compile the original Java code with the Java-TX-compiler as Java-TX is a conservative extension of Java. With this approach, it would be impossible to study the new features of Java-TX. Therefore, all type annotations are erased and the types are inferred by the Java-TX type inference algorithm.

**Infer new types and compare:** The inferred types may differ from the annotated types. We will compare the annotated and the inferred types. It is interesting to evaluate which type is more convenient. On the one hand, the inferred type could be more general such the code is better reusable. On the other hand, the inferred types could be less meaningful, such that the type annotations would help documenting the code.

**Maven for Java-TX:** Both Java and Java-TX can read classfiles. However, neither compiler can process source files of the other dialect. For the compilation of a project, where Java-TX- and Java-code is mixed (such as the Java-TX–Compiler in Java-TX), a make-tool is desired. This tool should be able to resolve dependencies between Java-TX and Java source files such that the files can be compiled in the correct order. Therefore we plan to write a maven plugin which can compile projects where Java and Java-TX code is mixed. With this approach however, circular dependencies between Java and Java-TX files cannot be resolved. It would be necessary for the Java-TX compiler to process Java source files to resolve such dependencies.

**First large project in Java-TX:** This is the first large project using Java-TX. Therefore, we will consider this project as a study to understand which pros and cons Java-TX has in comparison to original Java.

## 8 Unification algorithm as a webservice

The type unification algorithm [7] as a part of the type inference algorithm [8] is NP-hard [13]. Therefore, the run-time is often quite slow. The algorithm is qualified for parallelization, as different constraint sets are solved independently. Therefore the implementation is already parallelized [9].

We would like to offer the unification algorithm as a webservice on a powerful parallel system, such that complex type constraints can be solved even if the client does not have powerful hardware resources.

## 9 Module System for Java-TX

A new challenge is to transfer ideas from the complex SML module system to Java-TX.

One caveat of Java-TX is that — for performance reasons — only explicitly imported types are considered by type inference. That means, even commonly used types, such as `String` and `Integer` — which are implicitly imported in every Java program — are only considered by the type inference if `java.lang.String` and `java.lang.Integer` are explicitly imported, respectively.

That means, the developer must know which types are reasonable in the program and import them beforehand.

It would be desirable to develop a more powerful and efficient module system, than the current package and classpath based approach of Java.

For the sake of completeness, it should be mentioned that Java has had its own module system since version 9. However, it seems to address different problems than those arising in Java-TX. Specifically, it is used to combine multiple packages into modules and explicitly declare dependencies between them and protect internal code from external use.

SML's module system has some interesting aspects such as `signatures`, `structures` and `functors` in its module system.

However, in object oriented languages modularization, abstraction and code reusability is typically handled by classes and interfaces and namespace management is — at least in Java — primarily handled by packages.

It is ongoing work to evaluate if and how these aspects can be applied to Java-TX in a reasonable way.

## 10 Summary

In this paper we presented the Java-TX roadmap. Java-TX is a superset of Java, which extends the Java programming language primarily through a global type inference algorithm and real function types. There is an implementaiton of the compiler which realizes the fundamental features. But there is potential for improvement and extension. We showed possibilities to extend pattern matching, realizing Java-TX-signatures and non-variable bounds of generics for an extended approach of generics, and the implementation of real function types. Furthermore, it is desired to extend Java-TX by heterogeneous translation and a new module system. Finally, we showed the idea to implement the type unification as a webservice and presented the project to implement a Java-TX-compiler in Java-TX itself.

## References


[1] James Gosling et al. *The Java® Language Specification*. Java SE 8. The Java series. Addison-Wesley, 2014.

[2] Daniel Holle. “Ein neuer Pattern Matching Ansatz in Java-TX”. In: *Tagungsband des Jahrestreffens 2025 der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“*. Ed. by Daniel Holle et al. INSIGHTS – Schriftenreihe der Fakultät Technik 01/2025. (to appear). 2025. URL: https://www.dhbw-stuttgart.de/forschung-transfer/technik/schriftenreihe-insights.

[3] Ruben Kraft and Martin Plümicke. “Ein Language-Server für Java-TX”. In: *23. Kolloquium Programmiersprachen und Grundlagen der Programmierung – Vorläufiger Tagungsband*. Ed. by Stefan Brunthaler. Universität der Bundeswehr München. Sept. 2025.

[4] Martin Odersky, Enno Runne, and Philip Wadler. “Two Ways to Bake Your Pizza – Translating Parameterised Types into Java”. In: *Proceedings of a Dagstuhl Seminar, Springer Lecture Notes in Computer Science* 1766 (2000), pp. 114–132.

[5] Martin Plümicke and Andreas Stadelmeier. "Introducing Scala-like Function Types into Java-TX". In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ManLang 2017. Prague, Czech Republic: ACM, 2017, pp. 23–34. ISBN: 978-1-4503-5340-3.

[6] Martin Plümicke. "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. Ed. by Annette Bieniusa, Markus Degen, and Stefan Wehr. Vol. 413. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2024, pp. 43–56.

[7] Martin Plümicke. "Java Type Unification with Wildcards". In: *Applications of Declarative Programming and Knowledge Management, 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*. Ed. by Dietmar Seipel, Michael Hanus, and Armin Wolf. Vol. 5437. Lecture Notes in Computer Science. Springer, 2007, pp. 223–240.

[8] Martin Plümicke. "More Type Inference in Java 8". In: *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*. Lecture Notes in Computer Science 8974 (2015). Ed. by Andrei Voronkov and Irina Virbitskaite, pp. 248–256.

[9] Martin Plümicke. "Optimization of the Java Type Unification". In: *Proceedings of the 37th Workshop on (Constraint) Logic Programming (WLP 2023)*. Ed. by Sibylle Schwarz and Mario Wenzel. 2023. URL: https://dbs.informatik.uni-halle.de/wlp2023/WLP2023_Pl%C3%BCmicke_Optimization%20of%20the%20Java%20Type%20Unification.pdf.

[10] Martin Plümicke and Daniel Holle. "Principal generics in Java–TX". In: *22. Kolloquium Programmiersprachen und Grundlagen der Programmierung*. Ed. by Thomas Noll and Ira Justus Fesefeldt. Technical report / Department of Computer Science. - Informatik AIB-2023-03. RWTH Aachenen. Sept. 2023, pp. 122–143.

[11] Mark Reinhold. *Project Lambda: Straw-Man Proposal*. Dec. 2009. URL: http://cr.openjdk.java.net/~mr/lambda/straw-man.

[12] Julian Schmidt and Martin Plümicke. "Java-TX Compiler in Java-TX". In: *Tagungsband des Jahrestreffens 2024 der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte"*. Ed. by Daniel Holle et al. INSIGHTS – Schriftenreihe der Fakultät Technik 02/2024. 2024, pp. 33–51. URL: https://www.dhbw-stuttgart.de/fileadmin/dateien/Forschung/Forschungsschwerpunkte_Technik/DHBW_Stuttgart_INSIGHTS_2_2024_Tagungsband_Jahrestreffen_GI-Fachgruppe_Programmiersprachen_und_Rechenkonzepte_2024.pdf.

[13] Andreas Stadelmeier, Martin Plümicke, and Peter Thiemann. "Global Type Inference for Featherweight Generic Java". In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Leibniz International Proceedings in Informatics (LIPIcs) 222 (2022). Ed. by Karim Ali and Jan Vitek, 28:1–28:27. ISSN: 1868-8969.

[14] Etienne Zink. *Heterogene Übersetzung echter Funktionstypen in Java-TX*. (in German). DHBW Stuttgart/Horb. Studienarbeit, 2022.

# IMPRESSUM

**Schriftenreihe INSIGHTS**
**Themenreihe Engineering INSIGHTS**

**Herausgeber:**
Fakultät Technik der
Dualen Hochschule Baden-Württemberg Stuttgart
Postfach 10 05 63, 70004 Stuttgart

**Prof. Dr.-Ing. Harald Mandel**
**Prorektor für Forschung, Transfer und Nachhaltigkeit & Dekan Fakultät Technik**
Lerchenstraße 1, 70174 Stuttgart

E-Mail: harald.mandel@dhbw-stuttgart.de
Tel.: +49 (0)711 1849-605

www.dhbw-stuttgart.de/technik/insights

**Umschlaggestaltung:** Kerstin Faißt
**Bildnachweis:** Gerd Altmann auf Pixabay
**ISSN 2193-9098**

© Daniel Holle, Prof. Dr. Jens Knoop, Prof. Dr. habil. Martin Plümicke, Prof. Dr. Peter Thiemann, Prof. Dr. habil. Baltasar Trancón y Widemann (Hrsg.), 2025
Alle Rechte vorbehalten. Der Inhalt dieser Publikation unterliegt dem deutschen Urheberrecht. Die Vervielfältigung, Bearbeitung, Verbreitung und jede Art der Verwertung außerhalb der Grenzen des Urheberrechtes bedürfen der schriftlichen Zustimmung der Autorinnen und Autoren und der Herausgeberin.

Der Inhalt der Publikation wurde mit größter Sorgfalt erstellt. Für die Richtigkeit, Vollständigkeit und Aktualität des Inhalts übernimmt der Herausgeber keine Gewähr.

ISSN 2193-9098

www.dhbw-stuttgart.de/technik/insights