TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR FOR COMPILER CONSTRUCTION

Fakultät Informatik Institut für Technische Informatik, Professur für Compilerbau

Diplomarbeit

Extensions and Improvements for the Parallel Particle Mesh Environment

Tobias Nett

August 10, 2016

Supervisor
Dr. Sven Karol

Supervising professors
Prof. Dr. Jeronimo Castrillon
Prof. Dr. Ivo Sbalzarini

Statement of authorship

I hereby certify that I have authored this thesisentitled *Extensions and Improvements for the Parallel Particle Mesh Environment* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. They were no additional persons involved in the spiritual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, August 10, 2016

Tobias Nett

TECHNISCHE UNIVERSITÄT DRESDEN

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Fakultät Informatik, Institut für Technische Informatik, Professur für Compilerbau

Task Description for Final Thesis (Diplomarbeit)

For:	**Tobias Nett**
Degree program:	Informatik (Diplom)
Matriculation number:	3663974
E-mail:	tobias.nett@tu-dresden.de
Topic:	**Extensions and Improvements for the Parallel Particle Mesh Environment**

Domain-specific languages (DSLs) are of utmost importance in scientific high-performance computing to reduce development costs, raise the level of abstraction and, thus, make scientific programmer's life easier. The *parallel particle-mesh environment* (PPME) is a DSL and projectional editor for numerical simulations based on the particle method. PPME implements a generative approach: it generates parallel Fortran code that links with the *parallel particle-mesh library* (PPM), which is also implemented in Fortran. PPM provides efficient implementations of the particle and mesh abstractions, discrete numerics, as well as an abstraction layer on the underlying HPC hardware.

In its current state, PPME supports built-in abstractions such as particles, properties, fields, loops and computation phases. Moreover, systems of partial differential equations, differential operators such as the laplacian and fractals can be written using conventional mathematical notations. These concepts have been developed and tested using the example of a Gray-Scott reaction diffusion system, which is discretized and simulated using particles. However, while this example greatly shows the potentials of PPME w.r.t. particle-based simulations, it only includes a small set of equations and only two simulation phases—*initialization* and *solve*. Moreover, the editor lacks proper analysis features such as, for instance, type analysis and dead-code analysis. Also, potential optimizations to improve the efficiency of generated code or user experience were not considered yet.

This thesis therefore addresses these problems by extending and improving the PPME. In detail, it has the following goals:

- investigation and integration of additional simulation example(s) with multiple move and solve phases (e.g., Lennard-Jones),
- extension of PPME as needed to support the example(s),
- design and implementation of a type system (with support for physical units) and pending analyses,
- statically improve floating-point accuracy using abstract program equivalence graphs and
- provide an evaluation of the implemented optimization(s).

It is presumed that the student carefully analyses related work and tests his/her code using an appropriate testing methodology.

Start:	01.12.2015
End:	10.05.2016
1st referee:	Prof. Dr. Jeronimo Castrillon
2nd referee:	Prof. Dr. Ivo Sbalzarini
Supervisor:	Dr. Sven Karol

Prof. Dr. Jeronimo Castrillon
(Professor in charge)

Contents

1. Introduction

Numerical simulations have a become an essential part of science, alongside theory and experiments. As their implementation requires in-depth knowledge about the underlying models, numerical methods, and high-performance computing, the requirements on a researcher's programming skills become more and more demanding. Making scientific simulations accessible to a broad community — by providing state-of-the-art libraries and frameworks, offering high levels of abstractions, and reducing implementation costs and effort — is an ongoing task in the scientific computing community.

Domain-specific languages (DSLs) are of utmost importance to reach this goal by reducing development costs and raising the level of abstraction for simulation programmers. As the field of language-oriented programming has advanced in the last years, new tools for the design and implementation of DSLs have emerged.

The Parallel Particle-Mesh library (PPM) [Sba+06] is a framework for writing high-performance simulations based on particle methods. The Parallel Particle-Mesh Environment (PPME)[1] is a DSL and projectional editor built for this framework. It uses a generative approach to produce parallel Fortran code linked against PPM.

1.1. Motivation

PPME was developed as a research project and, in its current state, supports built-in abstractions for particle-mesh methods. The existing prototype provides language abstractions for particles, properties, fields, and loops, as well as systems of partial differential equations and differential operators. The notation of these concepts is kept close to the idiom of the domain, letting developers write differential equations in conventional mathematical syntax. The capabilities of PPME were demonstrated with an implementation of a Gray-Scott reaction-diffusion system, which was transfered from the original PPML. The model was discretized and simulated using particles.

While the early prototype of PPME shows the potentials of an integrated development environment for particle-based simulations, it is limited to a small set of domain abstractions, and usability has only been shown for a single use case. Moreover, the editor lacks proper features for analysis, such as type analysis and domain-specific checks. Although, domain knowledge can be used for potential optimizations to improve the efficiency of generated code, this aspect of domain-specific tooling is not considered in PPME yet.

[1] `https://bitbucket.org/ppme/ppme`

Thus, the objective of this thesis is to advance PPME to a full-fledged tool for scientific simulations based on particle methods. Therefore, PPME's feature set is expanded to enable the realization of new case studies. In order to simplify future extensions, a general framework for integrating external analysis tools is presented. The contributions of this thesis are an enabler for the productive use of PPME and open up the platform towards new case studies.

1.2. Contributions

The contribution of this thesis is three-fold. The existing prototype of the PPME is extended to cover common use cases of particle-mesh based simulations. The system is further improved by a complex type system and its implementation. Optional support for physical-unit annotation and the integration of external tools underlines the extensibility of the developed tool. Each of these points is addressed in the course of this thesis.

[C1] PPME Improvements and Case Studies The first contribution, as presented in Chapter 3, is the extension of PPME to support new case studies. At the same time, the applied changes prepare for extensions and plug-in solutions developed in this thesis. Therefore, this item is divided into two parts. The first part covers enhancement of PPME's feature set and internal measure of restructuring. Second, with the Lennard-Jones potential a second simulation example is transfered from the original PPML.

[C2] Types and Units The design and implementation of a formal type system with optional support for physical units is the second contribution. Chapter 4 presents the type system's conception and its implementation in detail. The second part of the chapter deals with the physical units extensions, which is an optional feature to harden simulations against unit errors.

[C3] Numerical Optimizations The final contribution concerns static optimization of floating-point expressions in PPME. Chapter 5 first presents an approach to numerical optimizations via program-equivalence graphs. Thereafter, a framework for the integration of external programs is developed and used to connect with Herbie[Pan+15], a tool for the optimization of floating-point expressions.

1.3. Organization

The remainder of this thesis is structured as follows. First background information for the further understanding of domain-specific languages language engineering is given in Chapter 2. In particular, the concept of language workbenches is explained. Furthermore, basic information on particle and particle-mesh methods is conveyed. The following three chapters treat the main contributions of this thesis.

Chapter 3 introduces the PPME development tool and points out specific features and details on the implementation. Two case studies are shown as examples of application. Subsequently, the second contribution in form of type system and physical unit extension is elaborated in Chapter 4. Hereafter, Chapter 5 discusses approaches on optimizing floating-point expressions. In addition, an external tool for the optimization of numerical expressions is integrated with the development environment. Finally, Chapter 6 evaluates the presented contributions and concludes the thesis with an outlook on future work.

2. Background

Writing fast and efficient code often requires not only in-depth knowledge about the problem domain, but also competence in the programming language of choice, the targeted systems (hard- and software prerequisites), and additional libraries. Domain-specific languages aim to offer an environment focused on the former while trying to hide the latter. In particular, this is an urgent issue in areas where more and more knowledge on many (unrelated) fields is required to achieve good results. The field of high-performance computing (HPC) with particle-based methods is affected by this *knowledge gap* [Sba09].

Section 2.1 explains the approach of *language-oriented programming* (LOP) [War94]. The focus is put on language workbenches as a tool for designing and implementing domain-specific languages. An in-depth presentation of the Metaprogramming System (MPS) [Dmi04] is given, as it forms the foundation for the contributions of this thesis.

Section 2.2 covers particle and particle-mesh methods for scientific models. The abstractions given by these methods allow to write simulations for a wide range of problem domains with a single framework. As an instance of a particle-mesh software toolkit the PPM library and the corresponding domain-specific language (PPML) [Awi+13] are presented.

2.1. Domain-specific Languages and Language Workbenches

The question what exactly a *domain-specific language* (DSL) is, is often subject to debates. DSLs even occur under different names, e. g., application-oriented [Sam69], special purpose [Wex81], or task-specific languages [Nar93]. All these names have in common that they emphasize the nature of the languages to be tailored for a specific task. In the following, the definition by Deursen et al. is used as a foundation [DKV00]. Note that the definition itself is still vague in terms of the problem domain.

> “A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. ”

Many languages are tailored for specific domains. To illustrate the variety in the DSL landscape, we name and categorize some domains and supplementing languages. One thing to be noticed is the diversity of fields of application. The range extends from the classical software engineering domain (such as financial products) over systems software (e. g., video-driver specifications) to telecommunication (e. g., communication protocols) [DK98]. Even the TEX macro language this document is written in is a DSL, specifically tailored for typesetting documents.

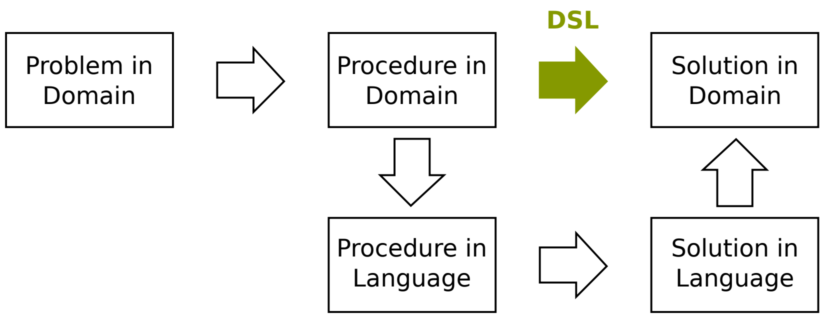


Figure 2.1.: DSLs allow for a direct implementation of a procedure defined in the problem domain. For an implementation in a general purpose language, the idiomatic procedure has to be translated to the language.

In contrast to general purpose languages (GPLs), DSLs allow to express solutions in an idiomatic way close to the level of abstraction of the problem domain. This enables a better communication between domain experts and developers, or even allows domain experts to become developers themselves. Figure 2.1 shows how the detour over a GPL can be avoided, because the procedure specification in the problem domain can directly be implemented in a DSL. Often, this means that solutions in a DSL are more concise than an implementation in a conventional GPL [Hud96]. Furthermore, studies have shown that DSLs can improve productivity and maintainability of software systems [DK98]. Most importantly, they embody domain knowledge, which allows for validation and optimization at the domain level, e. g., when processing catalytic chemical reaction networks [Hsu+08].

However, a DSL has to be designed, implemented, and maintained. These costs have to be taken into account when considering to introduce a domain-specific programming solution [DKV00]. Mernik et al. have investigated when to develop DSLs, addressing methodologies for their implementation [MHS05]. Moreover, availability of a DSL and corresponding compilers may be a concern. Distributing complete solutions, containing the DSL, compilers, and other tools, is one way to overcome this issue. The mbeddr system takes this approach [Voe+13].

All in all, a domain-specific language is tailored for solving particular problems or tasks while sacrificing expressiveness or usability for general problem solving. The key point for end users is the expressive power they get by using a DSL. By synchronizing the notations used in the DSL with common notations of the particular domain, DSLs can help to reduce the knowledge gap, i. e., the efficient use of computing resources requires more and more specialized knowledge and is thus restricted to a ever smaller community [Sba09].

2.1.1. Embedded DSLs

In general, two types of DSLs can be distinguished. On the one hand, there are *external DSLs*, which are standalone languages designed and implemented for a specific field of application. On the other hand, *embedded DSLs* are integrated in another programming language, lifting its expressiveness for a specific task. An external DSL is a newly defined language (often designed from scratch). For the new language both syntax and semantics can be defined freely, yielding great flexibility. However, developing a standalone language takes a lot of time and runs the risk of reinventing the wheel. Language designers have to balance these advantages and disadvantages. A prominent example is SQL [DD94], which is tailored specifically for the defining, managing, and querying relational databases.

In contrast to external DSLs, embedded DSLs (or *internal DSLs* [Fow10]) are built on top of an existing language. A language that ist syntactically embedded in a general purpose programming language offers both the specific language concepts related to the problem domain, and the ability to solve general problems by means of the GPL it is embedded in. Therefore, an embedded DSL does not restrict a programmer, but does provide means for idiomatic problem solving of specific aspects. Although embedded languages are often easier to develop, their syntax is restricted by the host language, and new concepts and functionalities have to adhere to the parent language.

Internal DSLs occur in several styles, as integrated mini-languages, or as language enhancements. The former refers to the deliberate decision of using only a particular subset of the host language for concise notations. The syntactic schema of the parent language is used to express the idiom of the domain. Often, the best notation for a domain has to be compromised to be conformant to the host language. This allows to provide a DSL as library, and existent tools and compilers can be used with the new language. Kossakowski et al. present an embedding of JavaScript in Scala, which enables safer client-side programming and more convenience by leveraging the host language's static type system and tools support [Kos+12]. The DSL is embedded using lightweight modular staging [RO12], a technique that allows to embed DSLs as libraries into host languages. Moreover, it enables domain-specific code generation, e. g., allowing to compile a program to a specific target language.

The latter style refers to the extension of a GPL with domain-specific concepts, e. g., by introducing new keywords. One way to implement language extensions is to pre-process the source code and translate the extensions to statements in the host language. The simplicity of this approach leads to some disadvantages when working with the DSL. For instance, users are provided with compiler feedback on the level of the parent language. Another option to implement embedded languages extensions are extensible compilers, such as ExtendJ [EH07; Öqv12] or Polyglot [NCM03]. The preprocessing phase is integrated in the compiler, which allows for type checking and optimization at the domain level.

2.1.2. Language Workbenches

A key issue with software, and software reuse in particular, is the gap between reusing concepts of problem solving and reusing the actual code [LR11b]. This discrepancy can be explained by the conceptional translation of abstract concepts into programming language patterns. For instance, the abstract concept of a state machine is often tangled with other concepts when implemented in a GPL. Closing this gap would not only make concepts behind implementations clearer, but it would also help to reduce the knowledge gap.

The use of DSLs, whether internal or external, can aid in this problem in that they allow programmers to code high-level abstractions directly. However, in order to be applicable in real-world software systems the implementation of DSLs has to be simple and fast, especially when one high-level construct touches another. Such interactions of DSLs can be handled by a common platform, which advocates their syntactic and semantic interoperability. This approach to software engineering is called *Language-Oriented*

Programming (LOP) [War94]. While the traditional approach for implementing a DSL using compiler-generator tools such as Lex and Yacc [LMB92] or ANTLR [Par13] can work for limited, fixed-size DSLs, it does not allow to define language concepts as separate components which can be reused and extended [LR11b].

Different LOP approaches have emerged to overcome this limitation, namely *LOP languages* and *language workbenches*, which both allow for flexible design and implementation of DSLs. *LOP Languages* like Cedalion [LR11a] are tailored to LOP [Ros10], similar to how object-oriented programming languages are oriented towards object-oriented programming (OOP). These languages are specifically designed to host internal DSLs.

Language workbenches are *integrated development environments* (IDE) that simplify the design and implementation of (domain-specific) languages [Fow05]. In addition, they allow for modularization and composition of languages [Erd+13]. Moreover, they provide some tooling like auto-completion and source-code navigation for free [LR11b]. In contrast to standard compiler-generator tools, language workbenches are specifically designed for language interoperability. Dmitriev, initiator of the *Meta Programming System* (MPS), describes language workbenches as a way to achieve independence and freedom to create, reuse, and modify languages and their environments [Dmi04].

This section shall give an overview of the language workbench concept, explain how languages are defined therein, and point out typical patterns for design and implementation. Erdweg et al. compare and discuss existing language workbenches in detail, capturing the design space of language workbenches in a feature model [Erd+13]. In the following, we outline the basic terminology of language workbenches.

One can differentiate two fundamental types of language workbench systems, *parser-based systems* and *projectional systems*. The two approaches differ in the way languages are implemented and handled.

Parser-based systems utilize grammars to describe the syntax of code written in the domain-specific language. A parser-generator is utilized to create a parser transforming a program into a data structure, e. g., into an *abstract syntax tree* (AST). Further analyses and transformations are performed on the AST and are usually applied before the final code generation. The Spoofax language workbench [KV10] is a parser-based platform for developing domain-specific languages and supplementing Eclipse editor plug-ins. Other textual workbenches such as JastAdd [SH11], EMFText [Hei+11] and Xtext [EB10] bring advances in editor and IDE technology to language engineering.

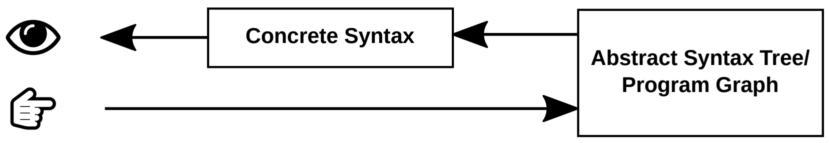


Figure 2.2.: In projectional editors, the user modifies the underlying model directly. The concrete syntax the user sees is projected from the program graph (cf. [Voe14, Figure 6.1]).

Projectional systems do not rely on parsers or parser generators, but the user modifies the abstract representation directly (cf. Figure 2.2). This approach is well-known from editing diagrams, where the user does not modify the pixels of the image, but rather handles geometric forms like rectangles as units. A projection engine provides one or more representations of the program graph with which the user can interact. This projectional editor (also structured editor [DHK84]) is a key factor for flexible representations. It allows to deal with arbitrary concrete syntactic forms, e. g., textual or graphical representations. Program elements in a projectional editor are stored as nodes with unique ID (UID).

References between elements are stored as references to the UIDs, lifting a plain AST to a program graph [Voe14]. The projectional editor establishes these references and aids developers with code completion. The Intentional Software Workbench [SCC06] is a commercial language workbench based on projectional editing. The view-based approach allows to mix graphical, tabular, and textual notations for programs. MPS is an open-source language workbench developed by JetBrains[1]. The tool supports language extension and modularization as well as different forms of concrete syntax [Voe11].

A language definition in a language workbench consists, in its core, of a *schema*, an *editor*, and a *generator*. These three parts define a language's metamodel and how it is presented, modified, and translated to a target language. The following paragraphs explain this in more detail.

Schema The schema defines the structure of the language and its elements, i. e., the *abstract syntax* or metamodel of the language. A schema definition therefore defines the abstract representation of program in the DSL. Typically, language concept is comprised of properties, child concepts, and references to other concepts. The definition of new language concepts usually starts with the schema definition.

To cite an example, a conditional statement consists at least of a boolean expression as condition and a body with statements to perform in case the condition evaluates to true. The common if-then-else construct, as used in many programming languages, further holds the optional child concept for the else-branch.

Editor The editor defines an element's presentation, i. e., the *concrete syntax* of the language. Thus, the editor defines the projectional view on the AST and in particular how the user can modify it. The technical details of editor specifications are subject to the language workbench instances, but usually they will access an element's properties, child concepts, and concept references and construct a textual or visual representation for the user.

Multiple editors for the same language concept can provide a different editing experience. A different representation of the abstract model is just a different projection, e. g., when showing the transitions of a state machine in a diagram or a table.

Generator The generator defines how the program graph is translated to an executable representation. Therefore, the generator defines the *semantics* of the DSL. The generator can be a compiler producing an executable binary, but often the target format is plain text or a base GPL such as Java, C, or Fortran. During the generation process, concept instances and their references can be analyzed and domain knowledge can be incorporated. Some language workbenches allow also for transformations between languages, e. g., from a high-level DSL to a lower-level base language.

Schema, editor, and generator are aspects of the language definition. Some workbench systems and frameworks have extended the set of language aspects for better separation of concerns and more flexibility. For instance, an attached type system simplifies type checking on the DSL and designated transformation descriptions allow to specify automatic code refactorings.

[1]`http://www.jetbrains.com/mps/`

JetBrains Meta-Programming System (MPS)

MPS is an instantiation of the language workbench concept. It offers a wide range of language aspects to adapt and customize the presentation and behavior of DSLs. Language elements are called *concepts* which are defined via different aspects. MPS has extended the pool of core aspects, consisting of structure, editor, and generator, by several new aspects, e. g., type system, dataflow, and find usages. For the specification of these aspects MPS provides a set of DSLs. Figure 2.3 gives an overview of the most important language aspects in MPS. Note that languages can be extended, and the generation process allows for transformations between languages. In the following, we present the main aspects involved in language definitions (cf. [Voe14; MPS15d] for more information).

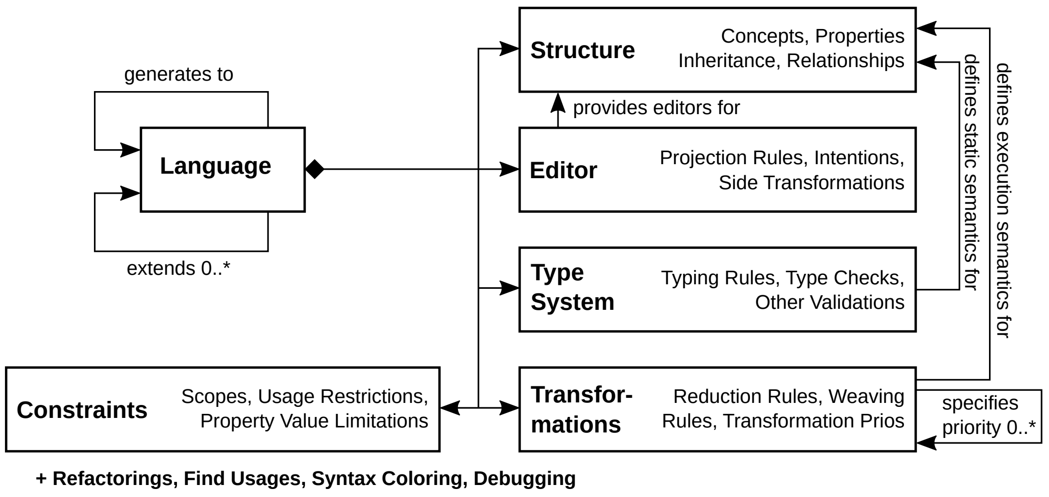

generates to
Language
extends 0..*
Structure
Concepts, Properties
Inheritance, Relationships
provides editors for
Editor
Projection Rules, Intentions,
Side Transformations
Type System
Typing Rules, Type Checks,
Other Validations
Constraints
Scopes, Usage Restrictions,
Property Value Limitations
Transfor-mations
Reduction Rules, Weaving
Rules, Transformation Prios
specifies priority 0..*
defines static semantics for
defines execution semantics for
+ Refactorings, Find Usages, Syntax Coloring, Debugging

Figure 2.3.: Languages in MPS are defined through several aspects (*structure* is used in MPS for the language schema). A language definition can be based and generate to another language. Such transformations can be provided with priorities, from which MPS calculates a transformation schedule automatically [Voe14, Figure 6.3].

Structure Structure is MPS' terminology for schema. A language concept in MPS consists of a name, primitive properties, child concepts, and references to other concepts. Similar to Java classes, structures can form hierarchies. A concept can implement any number of concept interfaces and it can extend up to one other concept.

Editor Editors in MPS consist of cells, where each cell can contain a constant (e. g., a keyword, symbol, or image), a property value of a concept, a child cell, or a reference. Furthermore, collection cells allow specific formatting for multiple elements. In addition, some editor reactions to specific user interactions in a given cell are defined in the editor aspect.

Transformations (Generators) Model transformations between languages in MPS are defined by generators. The optimal workflow is to lower high-level DSLs through transformations step-wise to a base language. Only the last stage of this multi-stage process performs a final translation to textual representation. A generator in MPS consists of a mapping script which contains transformation rules. A schedule for transformations is automatically created.

Textgen The textgen aspect for a concept defines the actual translation to an external target language. Ideally, this aspect is only used for concepts of a base language, such as `BaseLanguage`, the MPS implementation of Java. For higher-level concepts generators should be used as described above.

Type System The type system aspect allows to define typing rules for concepts. The rule set includes rules for type inference (e. g., the type of a variable reference is the type of the referenced variable), subtyping (e. g., `int` is a subtype of `float`), and checking rules to verify a model. A checking rule can evaluate any part of a model, e. g., to check for uniqueness of names or naming conventions.

Behavior The behavior aspect allows for custom methods on concepts, just like classes can hold methods in OOP. Such methods can be invoked on any instance of the concept in a polymorphic way. Hence, the concept instance can carry behavior in addition to their properties.

Structure and editor are sufficient to define a small, usable language in MPS. The textgen aspect enables the translation of the custom DSL to a target language. All other aspects are optional and improve the language's usability.

2.2. Parallel Particle-Mesh Methods

Modeling and simulation are essential parts of modern science. The hypothesis a model gives about a process or system can be used to answer questions about the system through simulations. Simulations show how a model behaves in specific situation, e. g., for specific parameter values. They are useful, or even necessary, when it is complicated or impossible to change these parameters directly in the model, e. g., the diffusion constant of a molecule cannot be changed. For this reason, simulations are well established in science, alongside theory and experiments [Sba09].

In scientific computing, particles can be used to simulate a variety of models computationally [Sba+06]. Particle methods are numerical schemes which can be used to simulate both discrete and continuous models. The basic mechanism of a particle-based simulation are particle interactions. When simulating a discrete model the simulation is called *item-based*, simulations of discretized continuous models are called *field-based*. This distinction is made to clarify the nature of the underlying model. Discrete models are naturally expressed with particles which directly represent discrete entities of the model, thus the term item-based simulation. Entities in the model can be atoms in molecular-dynamics or cars in traffic simulations, for instance. The properties these entities hold become particle properties, and particle interactions model the evolution of properties over time. Particles in field-based simulations do not directly correspond to entities in the model. Instead, the continuous fields to model are discretized, i. e., the field's values are only computed and stored at selected discretization points. Particles then correspond to these discretization points. Particle interactions are then discretized differential operators describing the continuous evolution of the model.

In any case, a particle is a point-like localized object to which several properties, such as volume, mass, velocity, or acceleration, can be ascribed. A particle's position can be in any space, and the properties can hold arbitrary information. The *evolution* of a particle denotes a change in its position and/or properties. Usually, the evolution is determined by pair-wise particle interactions.

Due to its generality, particle methods can be used for a variety of models. For instance, in molecular-dynamics simulations [FS01], particles represent atoms and the particle interaction describes the molecular force field. Smooth particle hydrodynamics [Mon92] is a particle method to simulate the field-based model of continuum fluid flow. The differential operators in the governing equations are discretized for the computation. Particle methods can also be used to solve image segmentation problems over field-based models,e. g., region competition [CPS12]. Here, particles mark boundary pixels between image regions.

Particle methods are designed for close-ranged particle interactions. An alternative way for evaluating long-range particle interactions are meshes. A *mesh* defines a grid laid over the simulation space and consists of a neighborhood relation of its cells. Furthermore, mesh cells can hold arbitrary properties. Thus, meshes define topological abstractions over the simulation domain, where the focus is to divide the domain in cells which can be computed effectively. The combination of particles and meshes form *hybrid particle-mesh methods*. A historic example for combined use of particles and meshes are plasma physics simulations [Hoc70].

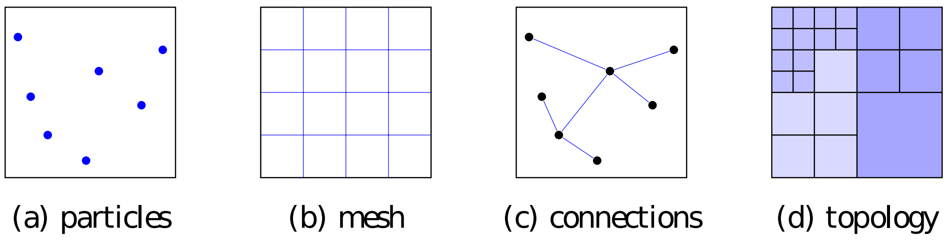


Figure 2.4.: Data abstractions in the PPM framework for particle-mesh simulations. Hybrid particle-mesh simulations can be expressed in terms of particles (a) and meshes (b). Connections (c) allow for associations of particles with each other, and a topology (d) decomposes the simulation domain [Awi+13, Figure 1].

The data and topology abstractions of hybrid particle-mesh methods can be utilized for highly parallel scientific computations [Sba+06]. Figure 2.4 presents a schematic view of these abstractions. Particle-particle connections enable to represent associations between particles, e. g., bonds in molecular dynamics. A domain decomposition given by a topology can be used to assign sub-domains to parallel processes for efficient computation. All in all, particle-mesh methods are a unifying framework for scientific simulations of both item-based and field-based models.

2.2.1. Parallel Particle Mesh Library (PPM)

The *Parallel Particle Mesh library* (PPM)[2] is a middleware layer which provides several abstractions for parallel hybrid particle-mesh simulations [Sba+06; BAS13]. In particular, PPM hides details about the underlying specifications of heterogeneous parallel hardware architectures without losing generality in terms of particle-mesh simulations. The current implementation is an object-oriented Fortran 2003 library which supports 2D and 3D models [ADS10]. By using standard technologies, languages (Fortran, C), and libraries (e. g., MPI [GLT99], METIS [KK98]) the library is portable and applicable on single processor machines as well as on high-performance computing clusters.

PPM is divided into two parts: *PPM core* and a *PPM numerics* library, as can be seen in Figure 2.5. The core library contains domain-specific routines and different modules for adaptive domain decomposition, communication through halo layers, load balancing, particle-mesh interpolations, and communication scheduling [Sba+06]. The numerical

[2] `http://www.ppm-library.org/`

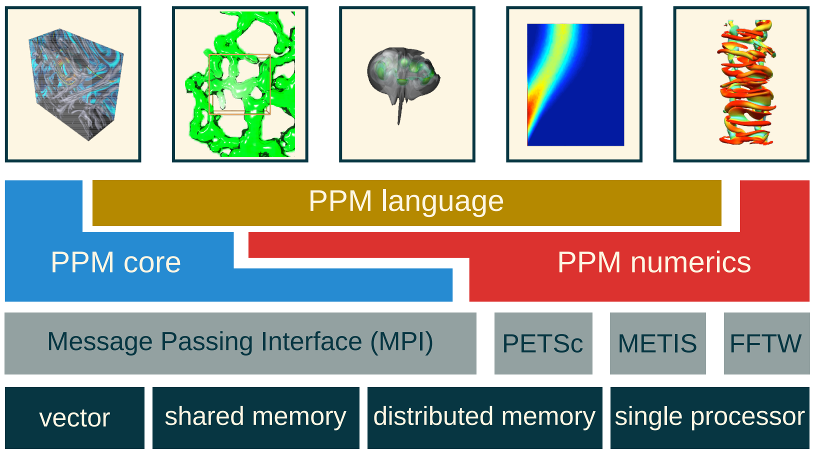


Figure 2.5.: The PPM software stack separates client programs from technical specifications. PPM consists of two encapsulated libraries, PPM core and PPM numerics. The numerics library uses routines provided by the core to implement numerical solvers. The domain-specific PPM Language aims to provide simpler access to particle-mesh simulations (cf. [Sba09, Figure 2]).

library supplements the core routines with frequently used numerical methods, e. g., mesh-based solvers, evaluation procedures for differential operators, and multi-stage ODE integrators [ADS10].

In addition to these modules, PPM provides bindings for some of the external libraries in case custom adjustments to simulations have to be made. By hiding technical details and deliberately focusing on the abstractions of particle-mesh methods, the PPM software stack provides a handy toolkit for scientific simulations.

2.2.2. Parallel Particle-Mesh Language (PPML)

The *PPM Language* (PPML) is a top-level abstraction layer for writing particle-mesh-based simulation code, i. e., a DSL for particle and particle-mesh simulations, located on top of the core library layer. It extends the PPM toolkit with a concise set of language abstractions reflecting the idiom of particle-mesh abstractions and a corresponding compiler.

The domain-specific language is “softly embedded” in Fortran, i. e., it syntactically forms an extension of the Fortran language. Technically, it is implemented by staged macro processing, transforming PPML client code to basic Fortran. This means that PPML code is not valid Fortran code but has to be (pre-) processed by the PPML compiler (or PPML preprocessor). In particular, the additional language constructs are macros which are statically expanded by the preprocessor to standard Fortran 2003 code. In a final step, the client code has to be compiled and linked to PPM. The language is located on top of the PPM architecture stack [Awi+13].

Figure 2.6 illustrate how a PPML client application is transformed to an executable program. The PPML compiler is build with the ANTLR parser generator and scans the client code for domain-specific concepts. The macro collection contains templates that are evaluated during compilation of the client. All data abstractions and convenience macros for frequently tasks (e. g., iterating over a particle list) are translated to Fortran code. The compiler and language design based on macros allows to extend PPML by simply adding new macros to the collection [Awi+13].

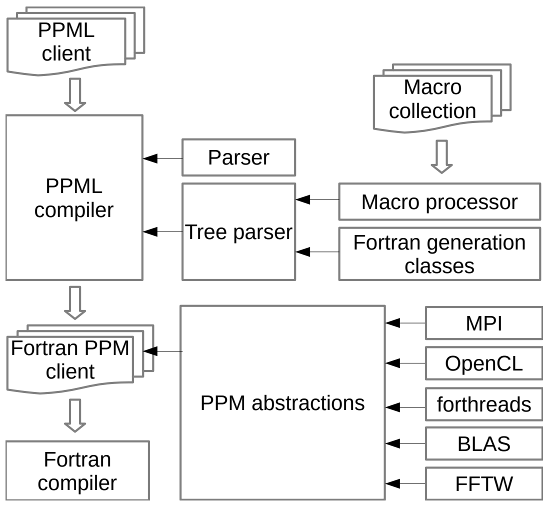


Figure 2.6.: The PPML framework and its compilation process. Line arrows denote usages, e. g., the PPML compiler a generated parser and the processed macros. Hollow arrows represent processing steps. The processing starts with a PPML client which is expanded by the PPML compiler using the macro collection to Fortran 2003 code [Awi+13, Figure 3].

PPML as an access layer to PPM helps to reduce the knowledge gap for writing scientific simulations with particle-mesh methods for high-performance computing. This is achieved by providing concise domain-specific instructions that allow to develop applications close to the idiom of the domain. Furthermore, full expressiveness is guaranteed as developers can fall back to Fortran as host language. Extensibility of the language is given through the open macro collection which allows to add new instructions and operations.

The main drawback of the current implementation of PPML is its lack of transparency. The staged macro expansion process hides details and semantic connections between instructions which may not be clear to the user. Moreover, the PPML compiler does not perform semantic checks on the source but solely relies on the syntactic structure of the macros. Thus, compilation errors are not reported until the Fortran compilation, and all errors are reported in the host language instead of the the PPML code. These error message are often of little help for a developer working with the DSL front-end [Kar+15].

3. The PPM Environment (PPME)

The *PPM Environment* (PPME) is an IDE and DSL for developing numerical simulations using the parallel particle-mesh method. It aims to reduce the knowledge gap, that is, the mismatch between domain experts and the required expertise for an efficient use of HPC resources [Sba09]. Therefore, it provides high-level abstractions and notations that are well-known to domain experts and, thus, aligns with the class of problems relevant in particle-mesh-based simulations. The development environment integrates with the existing software stack of PPM and does not require any adaption in the underlying framework.

The overall development process of applications can be improved by DSLs and development tools with regard to several criteria. Namely, these criteria are *productivity, software quality, performance/accuracy*, and *forward scalability* [Voe+12]. The *productivity* of scientists (i. e., developers) can be increased by providing high-level abstractions for computational models, such that the developer is not bothered with details about the programming language or the underlying architecture. The goal is to make users feel comfortable working with the environment. Although *software quality* is a criterion which is hard to measure, it can be stated that an IDE can check for common errors up-front and present the developer with meaningful warnings and error messages without running a compiler. For instance, PPME performs type checks on-the-fly, highlighting type errors directly in the editor. Additionally, static program analysis paired with domain knowledge can be used to improve performance, accuracy, and/or efficiency of simulations. Depending on the underlying technology, third-party tools can be incorporated for various compile-time optimizations. This allows to reuse established tools for analysis and program transformation instead of reimplementing their features. Finally, *forward scalability* is guaranteed by instantiating a flexible language workbench. Modularity and extensibility of the language and editor design allow to easily adapt to changes in the PPM framework and add new (language) features [Rat+12].

This chapter shall give an overview of PPME, its structure, implementation, and functionality. Combined, the presented aspects form contribution [C1], improvements for PPME and the implementation of a new case study. Section 3.1 explains how the MPS language workbench is instantiated and aligned with the PPM framework. Section 3.2 breaks down the internal transformations and code generation phase. Section 3.3 presents two case studies for simulation programs written in PPME.

3.1. Architecture

PPME is located on top of the existing PPM stack as a new access layer and is thus part of an extended PPM stack. It integrates with the layered architecture, generating source code against the PPML, and therefore makes use of the established work flow. Figure 3.1 illustrates how PPME fits between the user's programs and the PPM middleware. Application developers interact with the development environment to write simulation programs and related configuration files. Furthermore, the IDE provides a simpler access to the PPM library than PPML. The purpose of PPML and PPM of hiding technical details, specific realizations, and the explicit target platform is preserved. In contrast to PPML's macro extensions to Fortran, the IDE offers a consistent syntax and semantically incorporates domain-specific elements into the language.

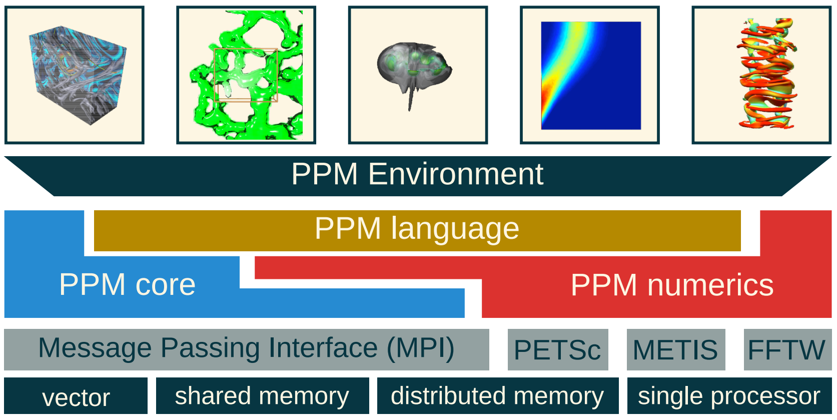


Figure 3.1.: PPME is a new access layer to the underlying PPM library for simulation developers [Sba09, Figure 2].

PPME itself is organized in language packages, called *solutions* in MPS. The clean separation between different sub-languages enables a good separation of concerns. The lower layers form a base DSL containing general language constructs such as expressions, literals, and statements. These concepts are reused in the upper layers to define domain-specific language concepts for particle-mesh methods on top of the base language. The reuse of lower layers and their extension is a key part in the design of PPME. It does not only allow for easier maintenance but also enables custom extensions for specific use cases as plug-ins to the IDE.

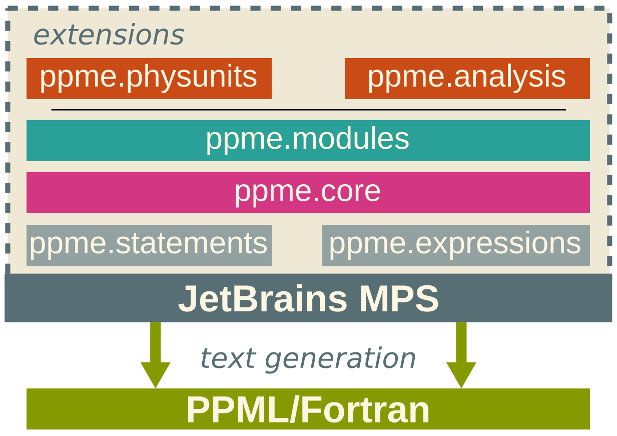


Figure 3.2.: PPME is built using a modular layered architecture approach.

Figure 3.2 shows a schematic view of PPME's language stack. The bottom layers form the main DSL for particle-mesh simulation programs that can be written in PPME. We use sub-languages to keep the implementation modular and understandable. At the interface to the underlying PPM library MPS is utilized to manage code transformations and code generation, as further explained in Section 3.2. The top-layer of PPME is open to extensions and allows to add application-specific languages. Such DSL extensions provide high-level abstractions that are only used in some domains. To cite an example, continuous and discrete simulations may use different constructs which are only meaningful in one domain. Another example could be particle-mesh methods for image processing [AS16]. Working with image data on particles may benefit from specific language features, e. g., for loading, storing, and visualizing images.

In the following, we present the contained sub-languages. A short characterization of their purpose should help the reader to get an understanding of PPME's design concept of modularity and extensibility.

`de.ppme.expressions` This solution provides general expressions as can be found in most programming languages, e. g., mathematical and logical expressions, and literals for integer and floating-point numbers. Moreover, the base types available in PPME are defined in this solution, as well as essential parts of the type system. Chapter 4 goes into more detail on the type system implementation.

`de.ppme.statements` The statements sub-language contains a basic set of imperative statements, such as expression statements, if-else clauses, and loops. Furthermore, variable declarations and references are part of this package. The type system of `de.ppme.expressions` is extended for variables where necessary. Overall, the elements of this language are general in the sense that they are independent of the domain they are used in.

`de.ppme.core` The core package is comprised of most domain-specific elements. It extends the solutions for expressions and statements by adding new domain-specific types, expressions, and statements. Selected constructs of PPML are modeled to be reflected in PPME while stying consistent with the base language's concepts. For instance, the `timeloop` construct available in PPM clients is implemented in this solution.

`de.ppme.modules` A module in PPME is the top-level structure for client programs written in PPME. It contains the simulation code and optional specifications for imported control parameters. A module corresponds to a PPM client, but the IDE can utilize additional knowledge about the domain better than PPML, e. g., by referencing external control files and inspecting the code.

The set of core languages of PPME are grouped in a *devkit*, a concept to treat a set of interconnected languages as one unit in MPS. To get the base functionality of PPME's language it suffices to include the devkit in a solution project.

`de.ppme.lang` The devkit containing the base languages of PPME. The developer does not have to worry about implementation dependencies of the core languages but can simply rely on the devkit. All language dependencies are covered by the devkit.

Above these central languages there is room for custom extensions and plug-ins. This work presents two additional languages, one for the integration of physical units into simulations, and one for the integration of an external analysis tool. Both should serve as examples for further extensions tailored for specific use cases.

`de.ppme.physunits` The physical units integration is an optional extension of the core DSL to integrate dimensional analysis capabilities in PPME. It adds means for unit specifications and a new type system layer. The language is presented in detail in Section 4.2.

`de.ppme.analysis` The analysis language consists of an exemplary binding of an external tool. We elaborate a general framework enabling the access of custom tools in the environment. Section 5.2 explains how Herbie [Pan+15], a tool for improving floating-point accuracy, is integrated in PPME.

The overall implementation effort for the current state of PPME was approximately eight person months. In the course of this thesis, the first IDE prototype was gradually enriched with new features and extended to cover additional case studies. This required major refactorings in the existing code base and various new implementations. Additionally, we investigated the integration of external tools and optional extensions in form of plug-ins.

3.2. Code Generation

The code generation in PPME is implemented by several intermediate transformation stages and a final text generation stage. A transformation maps one program to another. In the context of MPS, we usually work with transformations between different languages. A program specified using a high-level DSL is translated to a language closer to the target language. During the transformation, the program graph can be enriched with additional information which is explicitly required in the target language. For instance, a list of variables accessed in a loop can be extracted from the loop's body and explicitly stored in an intermediate representation.

The transformation concept allows for different fields of application. On the one hand, several transformations for the same model could be executed in parallel, yielding several representations of a single input program (e. g., producing customized versions for several target platforms). On the other hand, transformations can be chained, i. e., the result of one transformation is used as the input of the next one. Since PPME aims for modular language extension and composition, the underlying language workbench has to support transformations introduced by different language extensions. The transformation engine of MPS is able to resolve dependencies and compute a global transformation sequence for a given model. This allows to extend the language with new features and plug in transformations without changes to other mappings.

To avoid unnecessary overhead during the generation phase, e. g., producing textual representations of intermediate models, transformations are mostly limited to AST-to-AST mappings. Textual output is produced in only two cases: (a) when the final output in the target language is generated, and (b) when external tools and analyses require textual input. This restriction enables full control of the transformation phase within MPS. Taking into account the enrichments of various transformations and results of external tools, the platform is able to generate the final output in the target language. The produced source code can then be compiled using a regular compiler, i. e., the PPML compiler to create a Fortran binary of the simulation program.

Figure 3.3 illustrates this process. It starts with a simulation program implemented as a PPME module. The module contains domain-specific concepts such as a `timeloop` and various constructs to create and set up particle-based simulations. In the example of Figure 3.3, the `timeloop` statement is analyzed in the first transformation stage and a right-hand side specification is extracted. Similarly, the creation of particles is expanded to several initialization and macro calls in the representation of the module that is closer to the target language.

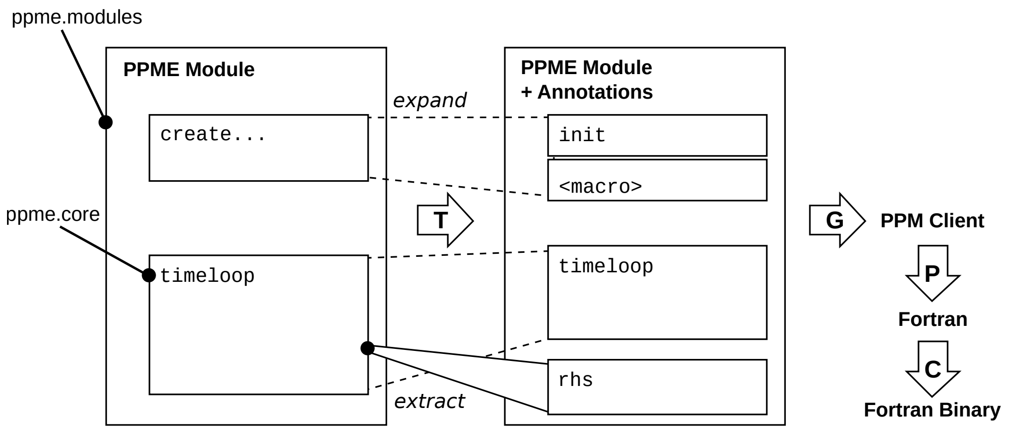


Figure 3.3.: Domain-specific abstractions in a PPME module are transformed (**T**) into lower-level representations. From this enriched module PPM client code is generated (**G**) which is subsequently processed (**P**) to Fortran code and compiled (**C**) to a Fortran binary.

In the following, a selection of transformations is presented. We elaborate how model transformations are used in PPME and how language extensions can add new mappings.

3.2.1. Main Transformations and Code Generation

The majority of the transformations is contained in `de.ppme.modules`, the top-level language of PPME which enables developers to write simulation clients. The mapping scripts are used to *pre-process* the input model, i. e., modify the source model by adding, removing, replacing, and changing nodes in the program tree. Pre-processing scripts are used for collecting certain information to be used in further transformations from the input model, or to perform non-template based model transformations. Mapping scripts have to be referenced from a *mapping configuration* in order to be invoked as part of its generation step [MPS15a]. A mapping configuration contains references to the pre-processing scripts to be invoked during the generation phase. Furthermore, it may contain other generator rules for template-based model transformations, reductions, or weaving.

```
mapping configuration   main
top-priority group      false

⋮

pre-processing scripts:
  collect_differentialOperators
  discretize_op
  populateRHS
  create_ode_macros
  copyUsedCtrl
  replaceRandoms
  populateAddArgMacros
  populateCreateFieldMacros
  populateCreatePropertyMacros
  transformForEachStatements
  addMKAnnotations

post-processing scripts:
  << ... >>
```

Listing 3.1: The mapping configuration of `de.ppme.modules`, containing references to pre-processing scripts to invoke non-template based model transformations.

Listing 3.1 shows the main mapping configuration used in PPME. A series of mapping scripts is referenced to be invoked during the generation step (ll. 25 – 46). In the following, we present three of them in detail, namely `replaceRandoms` (l. 41), `transformForEachStatements` (l. 45), and `populateRHS` (l. 38). Overall, the mapping scripts collect information from the input model and modify only specific parts of the model. Subsequent mapping scripts may use and reference the results of previous ones, e. g., the initialization of ODE macros requires a corresponding right-hand side to be present. The instantiation of several macros prepares for the code generation in the target language.

Generating Target Source Code In order to produce the target source code, MPS' *text gen* capabilities are utilized [MPS15b]. For each language concept a text generation component can be specified which creates a textual representation of the program node. The text generation composition can be as simple as printing the name of a variable (`VariableReference`), but it can also be more complicated, e. g., when producing the source code for a conditional loop statement.

We use several transformation scripts to prepare the text generation phase. Therefore, we have defined multiple intermediate concepts resembling macros in PPML, e. g., right-hand side definitions, or particle loops. The use of these intermediate representations simplifies the generation of PPML/Fortran code as the input model is stepwise enriched with the required information.

```
text gen component for concept PartLoopsMacro {
  (context, buffer, node)->void {
    indent buffer ;
    append {foreach } ${node.variable.name} { in } ;
    if (!node.pset.isInstanceOf(NeighborsExpression)) {
      append {particles(} ${node.pset} {)} ;
    } else {
      append ${node.pset} ;
    }
    append { with } ;
    append {positions(x} ;
    if (node.writePos) {
      append {, writex=} ${node.writePos + ""} ;
    }
    append {)} ;

    if (node.sca_field.isNotEmpty) {
      append modifiers node.sca_field, " sca_fields(", ")" ;
    }
    if (node.vec_field.isNotEmpty) {
      append modifiers node.vec_field, " vec_fields(", ")" ;
    }
    if (node.sca_props.isNotEmpty) {
      append modifiers node.sca_props, " sca_props(", ")" ;
    }
    if (node.vec_props.isNotEmpty) {
      append modifiers node.vec_props, " vec_props(", ")" ;
    }
    append \n ;

    with indent {
      append ${node.body} ;
    }

    indent buffer ;
    append {end foreach} \n ;
  }
}
```

Listing 3.2: Textgen specification for `PartLoopsMacro`.

Listing 3.2 shows the text generation component for the particle loop macro. MPS allows to print strings (`append`) and control the indentation of the output (`indent ↪ buffer` and `with indent`). We use the explicit information which is collected during the transformation phase and stored in the macro, e. g., the list of accessed fields and properties. The text generation for the loop's body is deferred to the respective program node.

Replacing Random Number Expressions The script entitled `replaceRandoms` is an example for a transformation script that analyzes the input model, adds new nodes, and replaces nodes of the input model. The script works on `RandomNumberExpression` (RNE), a concept offered by PPME to easily include random numbers in a simulation. The developer can choose the expression to be of a specific type, e. g., a random integer number `random<integer>`. If no type is specified, a random real number is assumed. During the code generation phase, the generic random number expressions have to be replaced by statements actually computing random numbers.

```
20    foreach e in exprs {
21      // keep track of variables already referenced in expression 'e'
22      nlist<VariableDeclarationStatement> used = new nlist<VariableDeclarationStatement>;
23
24      foreach rne in e.descendants<concept = RandomNumberExpression> {
25        node<LocalVariableDeclaration> varDecl;
26        node<VariableDeclarationStatement> stmnt;
27
28        // check whether a variable can be reused - if not, create a new variable declaration
29        if (rne.type.isNull) {
30          stmnt = decls.where({~it => it.variableDeclaration.type.isInstanceOf(RealType); }).disjunction(used).first;
31          varDecl = <LocalVariableDeclaration(
32            name: "rnd_" + cnt,
33            type: RealType())>;
34        } else {
35          stmnt = decls.where({~it => it.variableDeclaration.type == rne.type; }).disjunction(used).first;
36          varDecl = <LocalVariableDeclaration(
37            name: "rnd_" + cnt,
38            type: # rne.type)>;
39        }
40
41        // no variable for reuse was found - create a new declaration statement and add it to the list
42        if (stmnt.isNull) {
43          stmnt = <VariableDeclarationStatement(variableDeclaration: # varDecl)>;
44          decls.add(stmnt);
45          rne.containing root.descendants<concept = Phase>.first.body.statement.addFirst(stmnt);
46          cnt += 1;
47        }
48        // add the referenced variable to the list of 'used' variables
49        used.add(stmnt);
50
51        node<VariableReference> varRef = <VariableReference(variableDeclaration: # stmnt.variableDeclaration)>;
52        rne.replace with(varRef);
53        e.ancestor<concept = Statement>.add prev-sibling(<CallRandomNumber(var: # varRef)>);
54      }
55    }
56  }
```

Listing 3.3: Excerpt from the mapping script `replaceRandoms` to replace generic random number concepts with random number calls for local variables.

The script in Listing 3.3 iterates over each expression in a model containing RNEs and replaces them with references to variables holding a random number, reusing variables where possible. First, the script checks whether a variable to reuse can be found (ll. 29–39). If no variable was found, a new variable declaration with a unique identifier depending on the RNE's type is created. The new variable declaration is added to the beginning of the program (ll. 42–47). In contrast to Fortran, the developer does not have to declare a variable for each random number at the beginning of a procedure. Finally, the original expression is replaced by a reference to this variable (ll. 51–52). Since the abstract concept of a random number expression is lowered by the transformation to match the target language better, the random number is initialized by a call to Fortran's `RANDOM_NUMBER`, modeled by `CallRandomNumber`. Therefore, this call is inserted right before the expression containing the random number (`rootExpr`) (l. 53). Thus, for each random number expression a new variable is introduced which is initialized right before the expression it occurs in. Figure 3.4 illustrates this transformation. The random number expression is replaced by a variable reference to `rnd_1`, and a random value is assigned to the variable right before the expression statement via the `CallRandomNumber` instance.

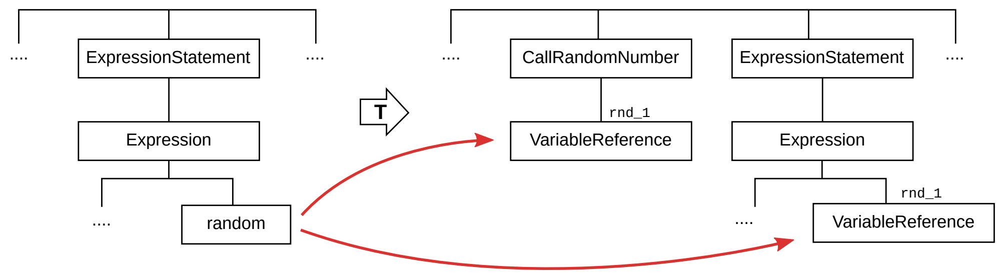


Figure 3.4.: Program-graph transformation translating random number expressions to variable references (red arrows).

Transforming Particle Loops The mapping script `transformForEachStatements` exemplarily shows how implicit information is extracted from a particle loop and then is encoded in a lower-level concept in an explicit form. PPML offers a convenient macro for iterating over each particle in a particle list. This macro requires explicit information about the accessed fields and properties of a particle. Furthermore, it has to be explicitly stated if particle positions are modified in the loop. The transformation script extracts this information from PPME particle loops and instantiates a corresponding intermediate concept `PartLoopsMacro`, modeling the PPML macro in PPME.

```
10      nlist<AssignmentExpression> assignments = particleLoop.body.descendants<concept = AssignmentExpression>.toList;
11      nlist<Expression> lhs = assignments.left.toList;
12      boolean writeX = lhs.findFirst({~it =>
13        if (it.isInstanceOf(ArrowExpression) && it : ArrowExpression.operation.isInstanceOf(PositionMemberAccess)) {
14          if (it : ArrowExpression.operand.isInstanceOf(VariableReference)) {
15            return it : ArrowExpression.operand : VariableReference.variableDeclaration == particleLoop.variable;
16          }
17        }
18        return false;
19      }).isNotNull;
```

⋮

```
28      particleMemberAccesses
29        .where({~it => it.operand.isInstanceOf(VariableReference); })
30        .where({~it => it.operand : VariableReference.variableDeclaration == particleLoop.variable; })
31        .forEach({~it => {
32          node<IVariableDeclaration> decl = it.operation : ParticleMemberAccess.decl;
33          node<VariableReference> ref = new node<VariableReference>();
34          ref.variableDeclaration.set(decl);
35
36          concept switch (decl.type) {
37            subconcept of FieldType :
38              if (decl.type : FieldType.ndim == 1) {
39                if (loop.sca_field.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.
                       ↪ variableDeclaration == ref.variableDeclaration; }).isNull) {
40                  loop.sca_field.add(ref);
41                }
42              } else {
43                if (loop.vec_field.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.
                       ↪ variableDeclaration == ref.variableDeclaration; }).isNull) {
44                  loop.vec_field.add(ref);
45                }
46              }
47              skip;
```

Listing 3.4: Excerpt from the mapping script `transformForEachStatements` to transform particle loop statements into particle loop macros.

Listing 3.4 depicts two key parts of the mapping script. First, by analyzing the original loop body the mapping script determines whether particle positions are written (ll. 12–18). Secondly, the script determines the accessed fields and properties in the loop's body. Particle member accesses are sorted into scalar/vector field/property usages. The listing shows the case distinction for particle fields (ll. 36–47). Finally, the original loop body is copied over to the macro, and the original loop is replaced by the created macro instance.

Overall, the slim and concise notation of a particle loop in PPME is enriched through the mapping with information implicitly available in the program. Since the full program graph is available to the mapping scripts, analyzing types of and dependencies between variables becomes easier. Access operations to particles are represented by specific language concepts and can be collected from the loop's body. Furthermore, complications when parsing the program for string-based macro processing are avoided. The comparison of the original loop in PPME and the generated macro code in PPML in Listing 3.5 reveals the collected information on the accessed properties. Since the particle position is written (`p→pos` occurs on the left-hand side of an assignment), the `position` is annotated with `writex=true`.

```
foreach particle p in parts do
  p→a = p→F / mass
  p→pos = p→pos + p→v * delta_t + 0.5 *
      ↪  p→a * delta_t²
  p→F = 0.0
  p→E = 0.0
end foreach
```

```
foreach p in particles(parts) with positions(x, writex=true)
     ↪ sca_props(E) vec_props(F, a , v)
  a_p(:) = F_p(:) / mass
  x_p(:) = x_p(:) + v_p(:) * delta_t + 0.5 * a_p(:) * (delta_t**2)
  F_p(:) = 0.0
  E_p = 0.0
end foreach
```

Listing 3.5: Comparison of particle loop in PPME (left) and in PPML (right).

Extract Right-hand Side Specifications The third transformation we present in detail is `populateRHS` which is responsible for extracting right-hand side definitions from ODE statements. A right-hand side specification in PPML contains directives for the computation of an ODE over a particle list. Our goal is to extract the corresponding code from differential equations written as part of a PPME simulation. Therefore, the script iterates over all `ODEStatement` blocks in a program and creates `RHSMacro` instances for each of them, where one ODE specification only works on one particle list `plist`. The key factor in extracting RHS definitions is to identify the differential operators in use and create a particle loop macro computing the differential equations.

```
25      // populate list of differential operators
26      result.diffops.clear;
27      ode.descendants<concept = DifferentialOperator>.forEach({~it => {
28          if (result.diffops.where({~elem => it.equals(elem); }).isEmpty) {
29            result.diffops.add(it.copy);
30          }
31        } });
32
33      // create field declarations for the used differential operators
34      result.diffops.forEach({~it => {
35          node<LocalVariableDeclaration> decl = new node<LocalVariableDeclaration>();
36          decl.name.set("d" + it.operand.operation : ParticleListMemberAccess.decl.name);
37          decl.type.set(it.operand.getType().copy);
38          result.appliedOps.add(decl);
39        } });
```

Listing 3.6: Extraction of differential operators from a PPME simulation. (`populateRHS`, ll. 25–39; Listing A.3)

Listing 3.6 shows how the distinct differential operators are assembled into a list. Within the loop over all ODE blocks, the occurrences of differential operators in the ODE block are collected. The function searching for descendants of a specific concept is conveniently provided by MPS, showing one advantage of the language workbench. Furthermore, a field declaration is created for each operator to store the intermediate result of applying the operator to a particle field in the macro instance. Finally, the declaration is added to the list of applied operators in the right-hand side macro. Note that the actual target source code is produced after all transformations have been executed, taking information from concepts for intermediate representation such as `RHSMacro`.

```
62        node<IVariableDeclaration> diffField = rhsStmnt.argument.operation : ParticleListMemberAccess.decl;
63        nlist<DifferentialOperator> diffops =
64          result.diffops.distinct
65            .where({~it => it.operand.operand.isInstanceOf(VariableReference) && it.operand.operand :
                   ↪ VariableReference.variableDeclaration == plist : VariableReference.variableDeclaration; })
66            .where({~it => it.operand.operation.isInstanceOf(ParticleListMemberAccess) && it.operand.operation :
                   ↪ ParticleListMemberAccess.decl == diffField; }).toList;
```

```
78        node<DifferentialOperator> diffop = diffops.first.copy;
79        diffop.operand.set(<ArrowExpression(
80          operand: VariableReference(variableDeclaration: # loop.variable),
81          operation: ParticleMemberAccess(decl: # rhsStmnt.argument.operation : ParticleListMemberAccess.decl))>);
```

Listing 3.7: Assembly of differential operators on particle attributes from `RHSStatement` (ll. 62–81; Listing A.3).

Next, the script scans all statements in the ODE block for right-hand side definitions (`RHSStatement`). The particle attribute affected by the operation is extracted from the statement as shown in line 62 of Listing 3.7. The subsequent statement selects the corresponding differential operator matching the equation, i. e., where the operand equals the particle attribute (ll. 64 ff.). Finally, the differential operator concept is set up with this information (ll. 79 ff.). Beforehand, the script assures that the applied differential operator is unambiguous. We assume that there is a single differential operator in the equation corresponding to the operand. Thus, the list of applied operators `diffops` should contain only a single element. The concrete differential operator used by the developer is assigned to the general concept of the partial differential equation. Note that the differential operator, specified over the particle list, works on the loop variable of the produced particle loop after the transformation.

Since the RHS statements in the ODE blocks of PPME operate on the particle list as a whole, all references to the particle list have to be transformed to references to a particle. Listing 3.8 shows how all these expressions are mapped to expressions over a single particle. Again, we utilize MPS' functionality to find descendants depicting particle list access and iterate over these occurrences (l. 87).

```
87        right.descendants<concept = ArrowExpression>.where({~it => it.operation.isInstanceOf(
              ↪ ParticleListMemberAccess); }).forEach({~it =>
88          node<> t;
89          if (it.operation : ParticleListMemberAccess.decl.isInstanceOf(VariableDeclaration)) {
90            t = it.operation : ParticleListMemberAccess.decl : VariableDeclaration.getType() ;
91          } else {
92            t = it.operation : ParticleListMemberAccess.decl.type : Type;
93          }
94
95          node<ArrowExpression> foo = it.replace with new(ArrowExpression);
96          foo.operand.set(<VariableReference(variableDeclaration: # loop.variable)>);
97          foo.operation.set(<ParticleMemberAccess(decl: # it.operation : ParticleListMemberAccess.decl)>);
98        });
```

Listing 3.8: Transformation of particle list accesses to direct particle accesses (ll. 87–98; Listing A.3).

Similar to the transformation of particle loops, the fields and properties used in the computation of the differential equations are extracted and explicitly added to the `RHSMacro` (cf. Listing 3.4). Finally, the `RHSMacro` is assembled using information about the particle list, the applied differential operators, and the equations itself. PPME's mechanisms ensure that the program is well-formed and that these transformations are applicable.

The comparison of the PPME ODE block (top) and the extracted right-hand side definition in PPML as shown in Listing 3.9 demonstrates the effect of the model transformation. The developer conveniently defines an ODE over attributes of the particle list `c`, and the IDE automatically extracts the required information. First, the mapping identified two applications of differential operators, $\nabla^2 \mathtt{c}{\rightarrow}\mathtt{U}$ and $\nabla^2 \mathtt{c}{\rightarrow}\mathtt{U}$. As mentioned before, the local variables dU and dV hold the intermediate result of applying the operator to the respective particle fields. Both the particle loop and the right-hands side block itself hold explicit information about the accessed particle fields, U and V. Furthermore, dU and dV are treated like other scalar fields. Note that the access of particle list attributes (c→U) is transformed to access of particle attributes, i. e., accessing attributes of the loop variable U_p.

```
ode method "rk4" on c
  ∂c→U/∂t = constDu * ∇² c→U - c→U * c→V² + F * (1.0 - c→U)
  ∂c→V/∂t = constDv * ∇² c→V + c→U * c→V² + (F + kRate) * c→V
end ode
```

```
rhs grayscott_rhs_0(U=>c, V)
  get_fields(dU, dV)

  dU = apply_op(L, U)
  dV = apply_op(L, V)

  foreach p in particles(c) with positions(x) sca_fields(U, V, dU, dV)
    dU_p = (((constDu * dU_p) - ((U_p * V_p) * V_p)) + ((1.0_mk - U_p) * F))
    dV_p = (((V_p * (U_p * V_p)) + (V_p * (-(kRate + F)))) + (constDv * dV_p))
  end foreach
end rhs
```

Listing 3.9: Comparison of and ODE specification in PPME (top) and the extracted right-hand side specification in PPML (bottom).

3.2.2. Physical Unit Transformations

PPME enables developers to enrich simulations with additional information about physical units of variables and constants. This functionality is provided by the language extension de.ppme.physunits which allows to attach unit information to declarations and expressions. The language is further explained in Section 4.2. However, the generation process does not know about the annotations. Therefore, we utilize MPS' capabilities to define new mapping scripts in a solution and add them to the generation plan. The IDE will sort out the order of mappings based on the languages involved.

de.ppme.physunits directly extends the main language of PPME by wrapping types and expressions with additional information about physical units, i. e., a type or expression can be replaced by a new annotated concept holding the original node and the annotation. The IDE uses this information to perform checks for consistency over the unit annotations. The developer gets visual feedback in the editor in case of errors. However, the target language cannot handle unit annotations wherefore they have to be removed from the model during the generation phase. Listing 3.10 shows the simple script replacing annotated types and expressions with the original type or expression, respectively.

```
mapping script removeUnitAnnotations

script kind     : pre-process input model
modifies model  : true

(genContext, model, operationContext)->void {
  foreach annotatedType in model.nodes(AnnotatedType) {
    annotatedType.replace with(annotatedType.primtype.copy);
  }
  foreach annotatedExpr in model.nodes(AnnotatedExpression) {
    annotatedExpr.replace with(annotatedExpr.expr.copy);
  }
}
```

Listing 3.10: Removal of annotated physical units from expressions and types.

We marked the generator priority of `de.ppme.physunits` to be higher than the generator of `de.ppme.modules` [MPS15d, Generator]. This ensures that the input model of the main transformation scripts will contain no annotated types or expressions. Note that this dependency is unidirectional where the extending language specifies an explicit dependency.

3.2.3. Transformations for external analysis

The integration of Herbie [Pan+15], a tool for optimizing floating-point expressions, allows to mark expressions in the editor for external analysis. PPME runs the analysis on the developer's request and annotates the marked expressions with optimized variants. We present the framework for integrating external tooling in detail in Chapter 5, including examples of simulations optimized by Herbie.

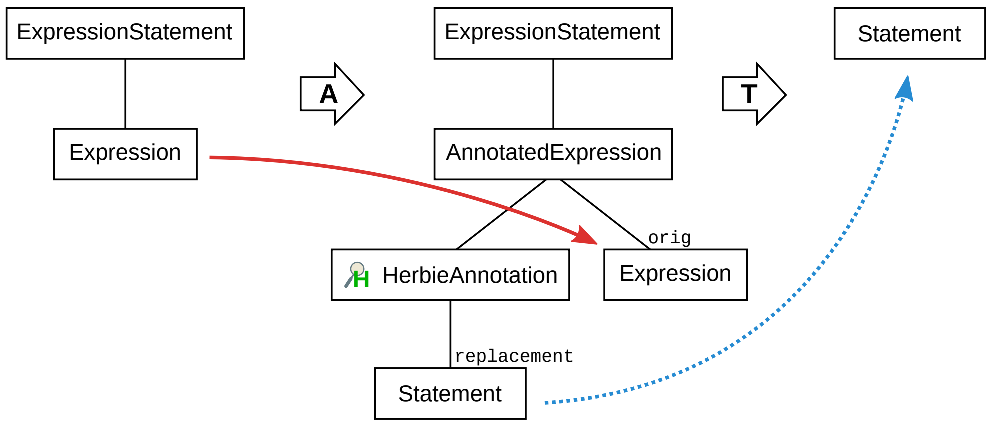


Figure 3.5.: A plain PPME expression is annotated (**A**) by the user and stored as `orig` in the annotation instance (red arrow). The `replacement` statement is created from the output of the external tool. During the transformation (**T**) the whole sub-tree is replaced by the optimized statement (blue dotted arrow).

During the code generation phase, annotated expressions have to be replaced by the annotated alternative (cf. Figure 3.5). A custom transformation script allows to add this functionality without modifying existing mappings. The script `HerbieOptimizations` from `de.ppme.analysis`, shown in Listing 3.11, iterates over all expressions with annotations. The parent statement of an annotated expression is then replaced by the optimized statement (i. e., the optimized *regime*) as found by the external tool. Note that the transformation does not check whether the replacement changes the program semantically. However, the replacement has to be a valid statement in PPME.

```
 7  foreach expr in model.nodes(AssignmentExpression).where({~it => it.@herbie != null; }) {
 8    if (expr.@herbie.replacement.isNotNull) {
 9      node<Statement> parentStmnt = expr.ancestor<concept = ExpressionStatement>;
10      if (parentStmnt.isNotNull) {
11        info "[Herbie Generator] replacing expression with optimized regime ...";
12        parentStmnt.replace with(expr.@herbie.replacement);
13      }
14    }
15  }
```

Listing 3.11: Expressions with annotated Herbie optimizations are replaced with the corresponding optimized regime by the `HerbieOptimizations` mapping script (ll. 8–15, Listing A.4).

As mentioned before, MPS prepares a generation plan based on dependencies between modules. The pre-processing script is classified to be run before the mapping scripts defined in `de.ppme.modules`. Thus, further transformations get a modified model as input where annotated expressions are replaced by their optimizations.

3.3. Case Studies

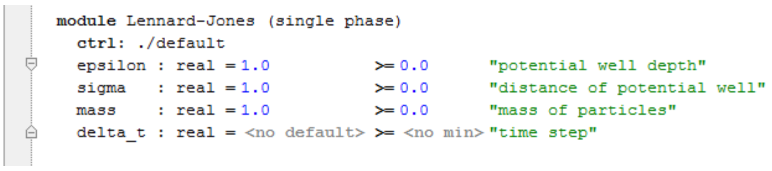
In order to demonstrate the capabilities of PPME we implemented two simulation case studies with the system. They exemplarily show the implementation of two kinds of models. First, the Lennard-Jones potential serves as an example for interacting particles, i. e., a discrete deterministic model. Second, the Gray-Scott reaction-diffusion system is implemented to show a continuous stochastic model of molecular dynamics with a low molecule number. In the following, we present both implementations in PPME and point out a few language features.

When simulating a discrete model, the simulation is called an *item-based simulation*, in contrast to simulating a continuous model which is called *field-based simulation*. These terms are used to clarify the nature of the underlying model, as the simulation itself is always discrete. For item-based simulations each item is represented by a particle in the simulation. As the name suggests, the particles model a field in field-based simulations, e. g., a concentration field. The diversity of the case studies underlines the flexibility of PPME and particle-based methods.

3.3.1. Lennard-Jones Potential

The implemented Lennard-Jones (LJ) potential [Ver67] is an instance of *molecular dynamics* [FS01], an item-based simulation to study atomic processes. The atoms subject to simulation are directly represented as particles, located in continuous space. Pairwise potentials between atoms define the continuous forces acting on them. While the basic algorithm for the simulation, i. e., computing pairwise interactions of particles and updating their positions and properties, stays the same, the exact definition of the forces is specific to the application. Thus, other use cases can be derived from the general scheme of the Lennard-Jones example.

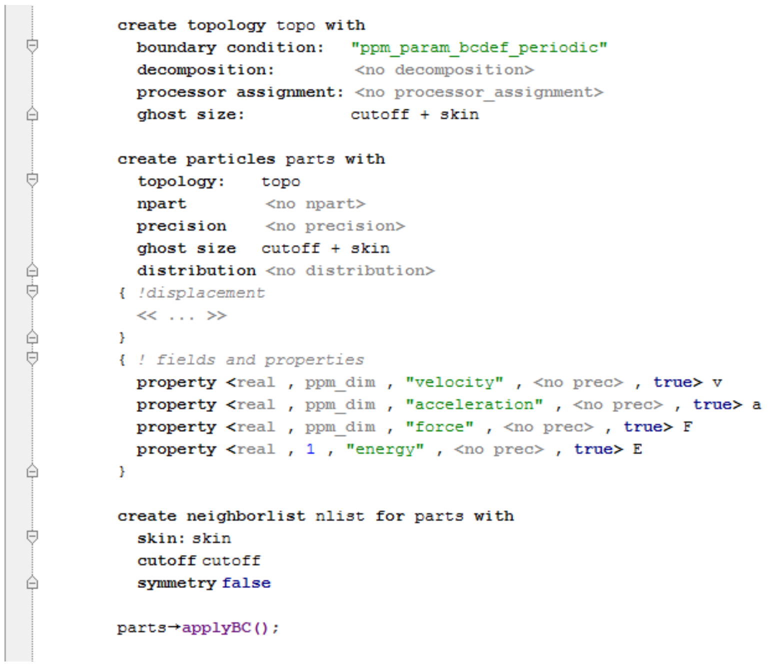
The implementation of the chosen case study is based on the PPML implementation. The pairwise potential of atoms is based on the distance between them (r), the depth of the potential well (ε), and the fall-off distance (σ). Particle properties such as acceleration (a) or velocity (v) are updated with regard to the potential and cause the particles to move. Additionally, a cutoff radius for negligibly small long-range interactions is applied. We go through the PPME simulation module and explain the key parts of the program in detail.



```
module Lennard-Jones (single phase)
  ctrl: ./default
  epsilon : real = 1.0          >= 0.0      "potential well depth"
  sigma   : real = 1.0          >= 0.0      "distance of potential well"
  mass    : real = 1.0          >= 0.0      "mass of particles"
  delta_t : real = <no default> >= <no min> "time step"
```

Figure 3.6.: **[LJ]** The module definition and referenced runtime constants of the Lennard-Jones potential simulation.

To begin with, the simulation is contained in a PPME module. Figure 3.6 shows the first lines of this module. The module references a control file (`ctrl`) which provides symbols for runtime constants. For each constant a default value, a minimal value, and a descriptive text can be supplied. For instance, the constant `epsilon` has a default value of 1.0, a minimal value of 0.0, and denotes the depth of the potential well in the model. Note that the exact value of these constants is not known at compile time but only on runtime of the simulation. Nevertheless, referencing a default configuration makes sure that all necessary symbols are defined and of the expected type. Besides the simulation client, the referenced control file is also generated.



```
create topology topo with
  boundary condition:   "ppm_param_bcdef_periodic"
  decomposition:        <no decomposition>
  processor assignment: <no processor_assignment>
  ghost size:           cutoff + skin

create particles parts with
  topology:    topo
  npart        <no npart>
  precision    <no precision>
  ghost size   cutoff + skin
  distribution <no distribution>
{ !displacement
  << ... >>
}
{ ! fields and properties
  property <real , ppm_dim , "velocity" , <no prec> , true> v
  property <real , ppm_dim , "acceleration" , <no prec> , true> a
  property <real , ppm_dim , "force" , <no prec> , true> F
  property <real , 1 , "energy" , <no prec> , true> E
}

create neighborlist nlist for parts with
  skin: skin
  cutoff cutoff
  symmetry false

parts→applyBC();
```

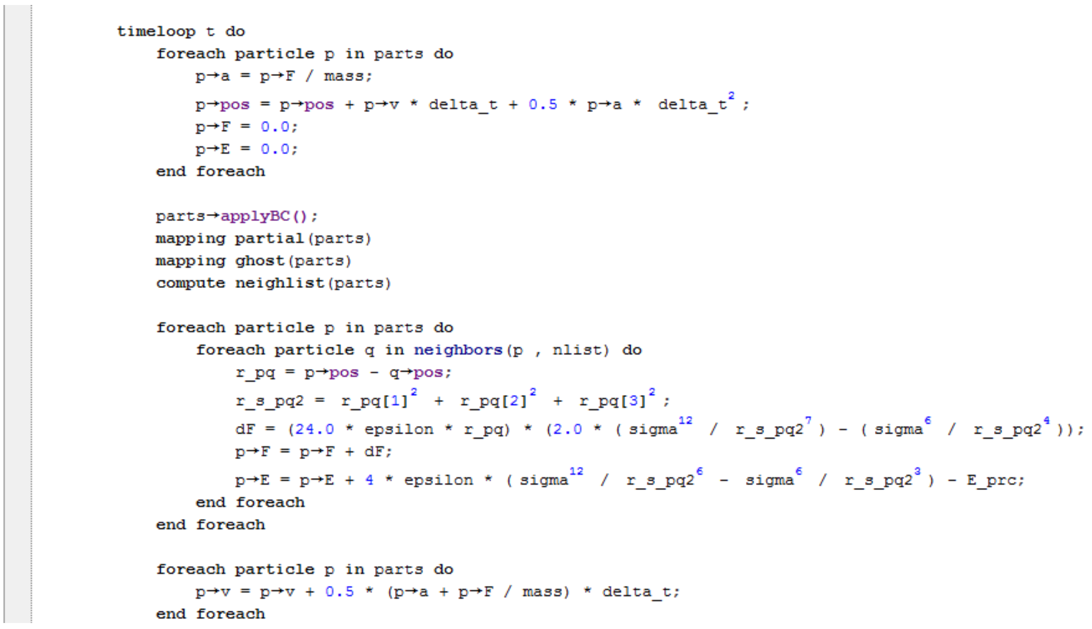
Figure 3.7.: **[LJ]** Initialization of topology, particles, and neighbor list.

The simulation body starts with a creation and initialization phase (cf. Figure 3.7). First, the topology for the simulation is created. In particular, we used a topology with periodic boundary condition (`"ppm_param_bcdef_periodic"`). Next, the particles are created over this topology. PPME allows to define an initial distribution of the particles in the simulation space (`displacement`) which is not used in this example. However, four properties are declared on the particle list. If not further specified, the type of a property defaults to a real number, and its default dimension is the dimension of the simulation (`ppm_dim`). The developer should provide a descriptive name for each property, e. g., *velocity*. The boolean flag indicates whether the property can be zero. Thus, the property `E` is a scalar denoting the energy value of a particle as a real number. Finally, the neighbor list `nlist` over the particles is computed, and the boundary condition is applied to the particles.

The essential part of simulating the potential is located in the `timeloop` depicted in Figure 3.8. Therein, the force acting on the particle in form of pairwise interactions is computed and applied. The loop can be divided into four parts. First, the particle positions ($p{\rightarrow}pos$) are updated based on the values of velocity ($p{\rightarrow}v$) and acceleration ($p{\rightarrow}a$). Since the particles' positions have changed the boundary condition is applied again, and the mappings and neighbor list are updated afterwards.

The block of two nested particle loops contains the actual particle interaction. For each particle p the pairwise interaction with its neighbor particles q, retrieved via `neighbors(p, nlist)`, is computed. The force F acting between two particles and the potential (or energy) E are given by

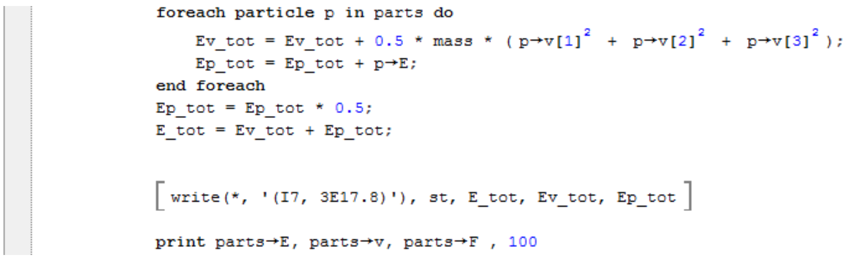
$$\vec{F}(r) = 24\varepsilon r(2\frac{\sigma^{12}}{r^7} - \frac{\sigma^6}{r^4}), \qquad V_{LJ}(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$



```
timeloop t do
    foreach particle p in parts do
        p→a = p→F / mass;
        p→pos = p→pos + p→v * delta_t + 0.5 * p→a *  delta_t^2 ;
        p→F = 0.0;
        p→E = 0.0;
    end foreach

    parts→applyBC();
    mapping partial(parts)
    mapping ghost(parts)
    compute neighlist(parts)

    foreach particle p in parts do
        foreach particle q in neighbors(p , nlist) do
            r_pq = p→pos - q→pos;
            r_s_pq2 =  r_pq[1]^2  +  r_pq[2]^2  +  r_pq[3]^2 ;
            dF = (24.0 * epsilon * r_pq) * (2.0 * ( sigma^12  /  r_s_pq2^7 ) - ( sigma^6  /  r_s_pq2^4 ));
            p→F = p→F + dF;
            p→E = p→E + 4 * epsilon * ( sigma^12  /  r_s_pq2^6  -  sigma^6  /  r_s_pq2^3 ) - E_prc;
        end foreach
    end foreach

    foreach particle p in parts do
        p→v = p→v + 0.5 * (p→a + p→F / mass) * delta_t;
    end foreach
```

Figure 3.8.: **[LJ]** The simulation loop computing pairwise interactions of particles.

The corresponding lines of code are the assignments to dF and $p{\to}E$, respectively. The last update on the particle list modifies the velocity with regard to the updated force field.



```
    foreach particle p in parts do
        Ev_tot = Ev_tot + 0.5 * mass * ( p→v[1]^2  +  p→v[2]^2  +  p→v[3]^2 );
        Ep_tot = Ep_tot + p→E;
    end foreach
    Ep_tot = Ep_tot * 0.5;
    E_tot = Ev_tot + Ep_tot;


    [ write(*, '(I7, 3E17.8)'), st, E_tot, Ev_tot, Ep_tot ]

    print parts→E, parts→v, parts→F , 100
```

Figure 3.9.: **[LJ]** A sanity check computing the total energy over all particles. Additionally, the intermediate simulation state is written out.

The final block shown in Figure 3.9 computes the total energy over all particles. The `print` statement generates an output file every 100 time steps, containing the values for the particles' energy, velocity, and force. The `write` directive, enclosed in square brackets, is copied directly to the generated source code. This enables developers to make adjustments using plain Fortran code.

The PPME module is translated to PPML client code using MPS' code generation capabilities. The simulation's output can be visualized with ParaView [Aya15], a data visualization tool. The particles are scattered in three-dimensional space, their movement governed by the potential. Figure 3.10 shoes a snapshot of the simulation. The particles are colored based on their current energy level.

3.3.2. Gray-Scott Reaction-Diffusion System

The second case study is the implementation of a reaction-diffusion model in PPME. Such continuous deterministic system models the time and space evolution of concentration fields of several species. The species react with each other and diffuse over

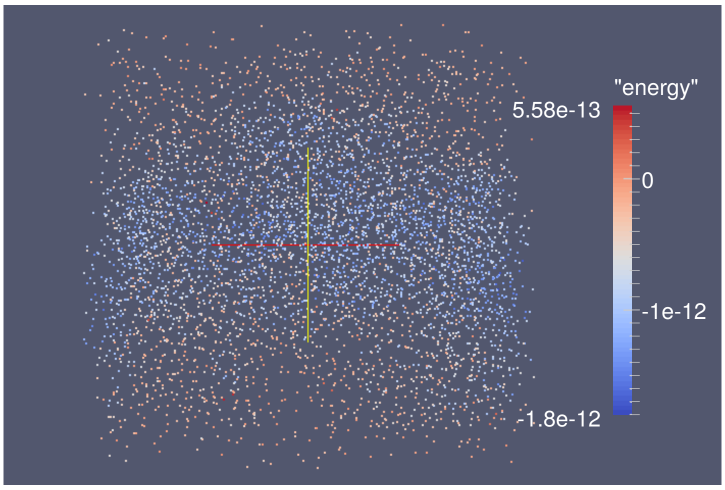


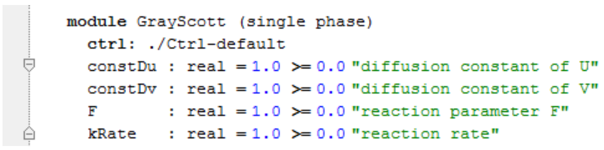
Figure 3.10.: **[LJ]** Visualization of the Lennard-Jones potential.

time [Awi+13]. The diffusion of a scalar quantity $U(x, t)$ is governed by partial differential equations $\partial U/\partial t = \Delta U$.

We implemented the Gray-Scott (GS) reaction-diffusion model [GS84] for two species U and V, based on the PPML implementation of the simulation. The reactions are modeled by the following differential equations

$$\frac{\partial U}{\partial t} = D_U \nabla^2 U - UV^2 + f(1-U), \qquad \frac{\partial V}{\partial t} = D_V \nabla^2 V - UV^2 + (f+k)V \quad (3.1)$$

where D_U and D_V are diffusion constants of species U and V, and f and k are reaction rates. Overall, the simulation can be divided into two sequential parts, first the initialization phase, and second the simulation loop. During the initialization phase, topology, particle list, and neighbor list are set up. The simulation loop simply contains a block of differential equations to model the evolution of the concentration fields. The notation is kept concise and close to the domain's idiom as most boilerplate code in the target language can be generated.

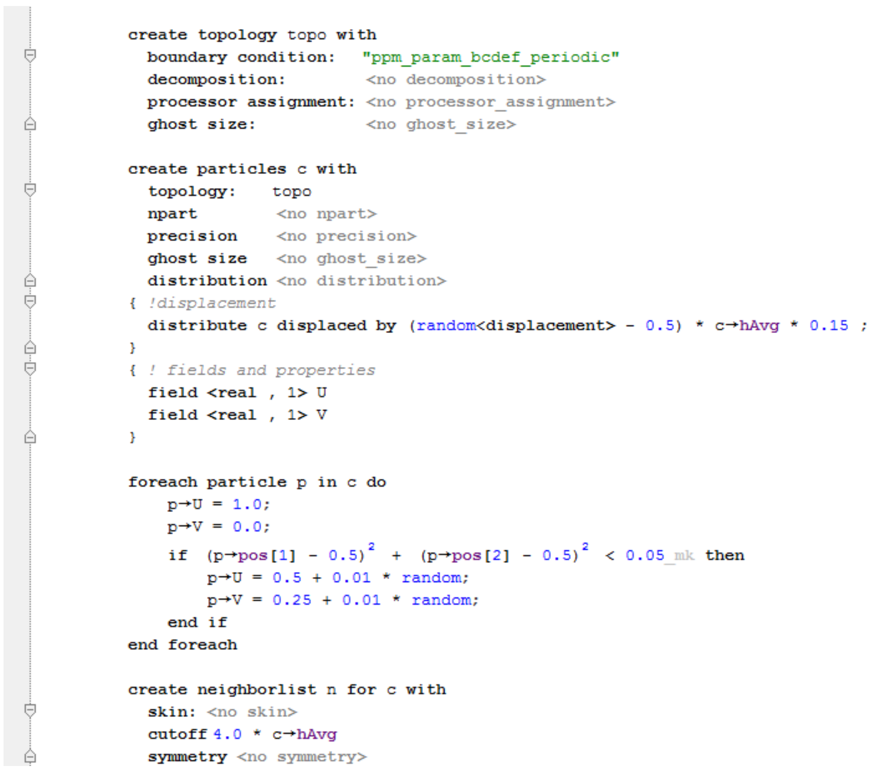


```
module GrayScott (single phase)
  ctrl: ./Ctrl-default
  constDu : real = 1.0 >= 0.0 "diffusion constant of U"
  constDv : real = 1.0 >= 0.0 "diffusion constant of V"
  F       : real = 1.0 >= 0.0 "reaction parameter F"
  kRate   : real = 1.0 >= 0.0 "reaction rate"
```

Figure 3.11.: **[GS]** Module definition and referenced runtime constants of the Gray-Scott reaction-diffusion system.

Similar to the Lennard-Jones case study, the reaction-diffusion system is implemented in a PPME module. Figure 3.11 shows the module definition and referenced runtime constants. Obviously, `constDu` and `constDv` correspond to the diffusion constants D_U and D_V, and `F` and `kRate` correspond to the reaction rates f and k, respectively.

Likewise the other case study, topology, particle list, and neighbor list are set up at the beginning of the simulation body. Figure 3.12 shows the directives to initialize the simulation environment. First, the topology is created, using the same boundary condition as for the Lennard-Jones potential. Then, the particle list `c` is created on the topology. This time, the displacement block is used to randomly distribute the particles. During the particle creation, scalar fields for the species U and V are defined. In the following particle loop the starting concentrations for the species are set up. Finally, the neighbor list is computed on the particles with a custom cutoff radius.



```
create topology topo with
  boundary condition:   "ppm_param_bcdef_periodic"
  decomposition:        <no decomposition>
  processor assignment: <no processor_assignment>
  ghost size:           <no ghost_size>

create particles c with
  topology:    topo
  npart        <no npart>
  precision    <no precision>
  ghost size   <no ghost_size>
  distribution <no distribution>
{ !displacement
  distribute c displaced by (random<displacement> - 0.5) * c→hAvg * 0.15 ;
}
{ ! fields and properties
  field <real , 1> U
  field <real , 1> V
}

foreach particle p in c do
    p→U = 1.0;
    p→V = 0.0;
    if  (p→pos[1] - 0.5)^2  +  (p→pos[2] - 0.5)^2  < 0.05_mk then
        p→U = 0.5 + 0.01 * random;
        p→V = 0.25 + 0.01 * random;
    end if
end foreach

create neighborlist n for c with
  skin: <no skin>
  cutoff 4.0 * c→hAvg
  symmetry <no symmetry>
```

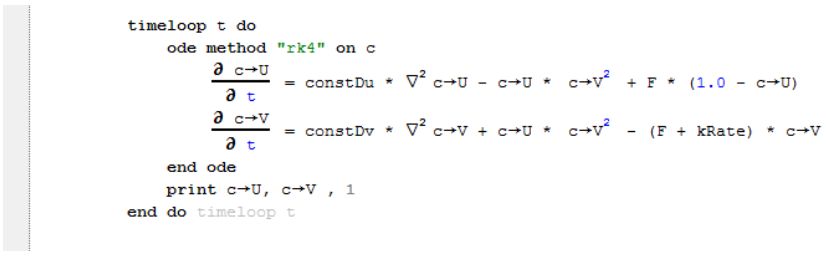
Figure 3.12.: **[GS]** Initialization of topology, particle list, and neighbor list. In addition, the species concentrations for U and V are modified.

```
22  c = create_particles(topo)
23
24  allocate(rnd_0(ppm_dim,c%Npart))
25  call random_number(rnd_0)
26  allocate(displacement(ppm_dim, c%Npart))
27  displacement = (rnd_0 - 0.5_mk) * c%h_avg * 0.15_mk
28  call c%move(displacement, info)
29
30  call c%apply_bc(info)
31  global_mapping(c, topo)
32
33  U = create_field(1, "U")
34  V = create_field(1, "V")
35  discretize(U, c)
36  discretize(V, c)
37
38  ghost_mapping(c)
```

Listing 3.12: Target source code generated from the particle creation statement.

Note that the continuous fields U and V are automatically discretized on the particle list during the code generation. Listing 3.12 shows the code snippet generated from the particle creation statement (`create particles`). Lines 24–28 correspond to the distribution directive, computing the displacement and applying it to the particle list. In lines 33–36 the continuous fields U and V are created and directly discretized on the particle list `c`.

The simulation logic is contained in the `timeloop` and is solely defined through differential equations. For the ODE block, the developer has to specify the particle list the equations are working on, and the method to approximate the solution of the ODE. The key `"rk4"` stands for the 4th order Runge-Kutta method, defined in the PPM library. The following two equations correspond to Equation (3.1). Note how closely the mathematical notation is resembled in PPME.



```
timeloop t do
    ode method "rk4" on c
        ∂ c→U
        ─────  = constDu * ∇^2 c→U - c→U * c→V^2 + F * (1.0 - c→U)
         ∂ t
        ∂ c→V
        ─────  = constDv * ∇^2 c→V + c→U * c→V^2 - (F + kRate) * c→V
         ∂ t
    end ode
    print c→U, c→V , 1
end do timeloop t
```

Figure 3.13.: **[GS]** Simulation loop containing the ODE describing the evolution of concentration fields U and V.

During the generation phase the simulation loop is analyzed and transformed into several macro instantiations in PPML. On the one hand, the code is scanned for differential operators in use which are set up before entering the loop. In this case, the Laplace operator `Lap` was discretized, as can be seen in Listing 3.13. Furthermore, a right-hand side definition is extracted from the ODE block as shown in Listing 3.9. The ODE is evaluated in each simulation step `nstages` times.

```
o_0, nstages_0 = create_ode([U, V], grayscott__single_phase__rhs_0, [U=>c, V], rk4)
L = discretize_op(Lap, c, ppm_param_op_dcpse, [order=>2, c=>1.0_mk])

t = timeloop()
  do istage=1, nstages_0
    ghost_mapping(c)
    ode_step(o_0, t, time_step, istage)
  end do
  print([U=>c, V=>c], 1)
end timeloop
```

Listing 3.13: Target source code generated from the simulation loop.

The output of the simluation was again visualized using ParaView. Figure 3.14 shows the concentration field of the species U and V after $4 \cdot 10^3$ time steps. In the initial concentration, a cicular spot of species V was surrounded by species U. Over time, this visible patterns formed out, based on the chosen paramters for D_u, D_v, f and k.

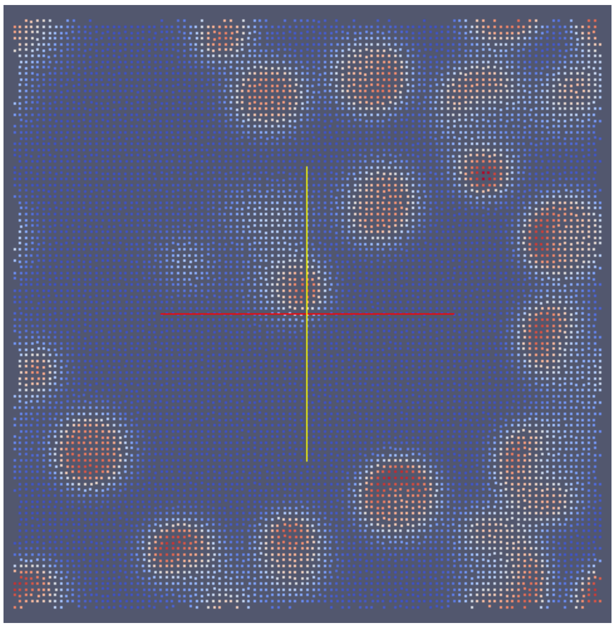


Figure 3.14.: **[GS]** Visualization of the Gray-Scott reaction-diffusion system.

4. Types and Units

An essential part of PPME is the implementation of a domain-specific language tailored for particle-mesh simulations and its static analysis of types. The type system does not only cover common aspects of programming languages but also includes type checks for constructs special to the given domain.

The work presented in this chapter covers contribution [C2]. A formal type system, its type hierarchy, typing rules, and error detection capabilities are presented in Section 4.1.

The second part of this chapter deals with the question how information on physical units can be added to PPME. Section 4.2 presents a unit calculus as a modular language extension. Unit annotations are an optional feature of the development environment, aiding development and debugging in a constructive way.

4.1. The PPME Type System

The DSL of PPME is backed up by a type system implementation using the language aspects provided by MPS [MPS15c; MPS14]. Making type system implementations easier is also a concern of other research. JastEMF by Bürger et al.[Bür+11] integrates the JastAdd system [EH07], a generator for reference attribute grammars, with the Eclipse Modeling Framework (EMF), a metamodeling framework. Heidenreich et al. present a textual DSL build with JastEMF, and Bürger et al. created SiPLE, an imperative programming language, with JastEMF and Xtext [Bür+11; Hei+11]. Bettini's *Xsemantics* is a DSL for writing type systems for Eclipse Xtext, and the Veritas workbench by Grewe et al. aims to provide a high-level specification language for type systems from which soundness proofs can be derived [Bet15; Gre+15]. MPS' typing language aspects head for the same goal. The type system is a key part constructively in improving the code quality of simulation written in PPME as it detects errors at compile time and gives meaningful feedback to the developer.

First, we introduce the notions and notations used to describe and formalize the type system in section 4.1.1. Second, we give an overview of the available types and operations in PPME, i. e., basic and domain-specific types and operations. The type-inference rules of PPME are elucidated (section 4.1.2) as well as sources of typing errors and how they are reported to the user (section 4.1.3). Finally, the technical implementation is presented in section 4.1.4.

Notions and Notations

We want to give some basic notational agreements and terminology for a general understanding of the type system. A *typing environment* Γ associates variable names and types, i. e., it is a set of pairs $\langle x, \tau \rangle$, commonly written as $x : \tau$, denoting that the variable x has type τ within the environment. We denote the lookup of a variable's type by $\Gamma(x)$, where $\Gamma(x) = \tau$ if the environment contains an entry for the variable $\langle x, \tau \rangle \in \Gamma$.

For some rules, the subtype relationship is of importance. If T and S are types, then $T \leq S$ denotes that T is a (direct) *subtype* of S, and S is a *supertype* of T. $T \leq^* S$ is the reflexive transitive closure of $\leq$, that is, S can be reached from T in the type hierarchy. In the remainder of this chapter, we use $\leq$ for simplicity and refer it to the transitive closure.

The inference rules for types and units are syntactically based on the representation in [Clé+86]. Each rule defines the conclusion that can be drawn from the given premises:

$$\frac{premise_1 \quad \ldots \quad premise_n}{conclusion}$$

We allow typings as premises as well as other predicates, e. g., for specifying a subtype relation.

4.1.1. Types in PPME

In MPS, any language element can function as a type in the internal type system. For the implementation of PPME's language a common super element `Type` was employed from which all other types are derived. As a whole, the type system in PPME can be seen as the combination of two parts — a general base type system on the one hand, and a domain-specific extension thereof on the other hand. Subsequently, the sets of basic and domain-specific data types of PPME are introduced. Afterwards, the inference rules of the type system are presented.

Base Types The base type system is defined in `de.ppme.expressions`. It consists of a set of fundamental types and type-inference rules over this set. The set $\mathcal{P}$ of elementary or primitive types is tailored to the domain of PPME. It contains the well-known types `String` for character sequences, `Boolean` for truth values, `Integer` for integers, and `Real` as an abstraction for the real numbers.

$$\mathcal{P} = \{String, Boolean, Real, Integer\}$$

Additionally, PPME knows two kinds of parameterized types for vectors and matrices, denoted by `Vector<X>` and `Matrix<X>` where `X` is a *type parameter*. A vector and matrix represent one-dimensional and two-dimensional arrays with components of type `X`, respectively. These polymorphic types are joined under an abstract `ContainerType` in the implementation and are denoted by the set $\mathcal{C}$.

$$\mathcal{C} = \{Vector\langle X \rangle, Matrix\langle X \rangle\}$$

The set of base types $\mathcal{T}_{Base}$ is then defined as the union of primitive types $\mathcal{P}$ and container types $\mathcal{C}$.

$$\mathcal{T}_{Base} = \mathcal{P} \cup \mathcal{C}$$

Figure 4.1 illustrates the hierarchy of types in `de.ppme.expressions`. In this case, `Type`, `Primitive`, and `Container<X>` are abstract types and do not occur directly in programs. Thus, the only case of subtyping is $Integer \leq Real$, expressing that every integer is also a real number. Furthermore, the container types $Vector\langle X \rangle$ and $Matrix\langle X \rangle$ are parameterized with a type parameter X which defines the container's component type. X can be of any type $t \in \mathcal{T}$ defined in PPME.

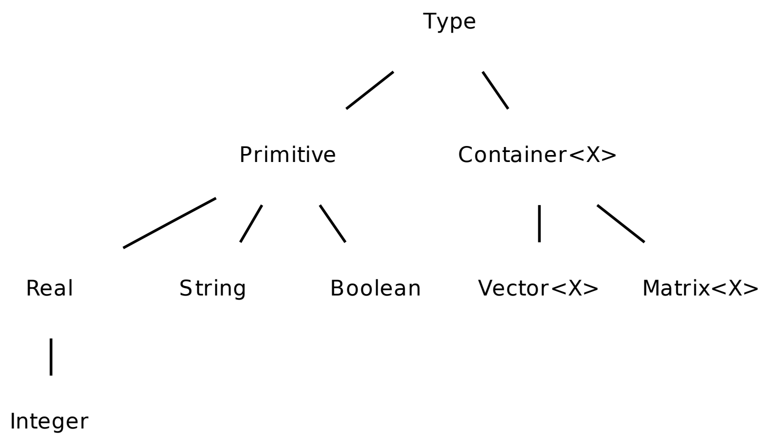


Figure 4.1.: The hierarchy of base types $\mathcal{T}_{Base}$ as defined in `de.ppme.` ↪ `expressions`.

Domain-specific Types Besides other domain-specific elements and language features, `de.ppme.core` contains domain-specific extensions to the base type system. On the one hand, there are specific types required by the particle-mesh domain, e. g., types for representing particles, particle lists, and different kinds of properties. Furthermore, type refinements for specific purposes are introduced, e. g., $Displacement$ as a kind of real-valued matrix describing a relocation of particles. Similarly, a $ParticleList$ is a particular vector with particle components. For this purpose, PPME provides a set of domain-specific types $\mathcal{D}$.

$$\mathcal{D} = \{Particle, ParicleList, Field, Property, Displacement\}$$

On the other hand, the underlying PPM framework allows for various configurations of the simulation environment. For instance, it enables the developer to decide on a specific boundary condition, e. g., a fixed or periodic boundary of the simulation space. PPME takes these kinds of types into account and models them respectively. The set $\mathcal{T}_{PPM}$ of domain-specific type extensions to $\mathcal{T}_{Base}$ then contains the domain-specific types of $\mathcal{D}$ joined with additional "configuration" types like $Topology$ and $Boundary$.

$$\mathcal{T}_{PPM} = \mathcal{D} \cup \{Topology, Boundary\}$$

In Figure 4.2 the extensions to $\mathcal{T}_{Base}$ are illustrated. The types of $\mathcal{T}_{Base}$ defined in `de.ppme.expressions` are denoted in gray. The hierarchy indicates the specialized subtyping for $Displacement \leq Matrix\langle Real\rangle$ and $ParticleList \leq Vector\langle Particle\rangle$. On the right, the domain-specific types for particles and their properties are shown.

PPME Types Altogether, the set $\mathcal{T}$ of all types in PPME is the union of base types and domain-specific types.

$$\mathcal{T} = \mathcal{T}_{Base} \cup \mathcal{T}_{PPM}$$

The construction of $\mathcal{T}$ also shows the flexibility of language implementations in MPS. The fundamental type hierarchy can be extended by by new languages, adding new domain-specific concepts. Furthermore, this enables a continuous development and refinement of the given implementation towards new use cases.

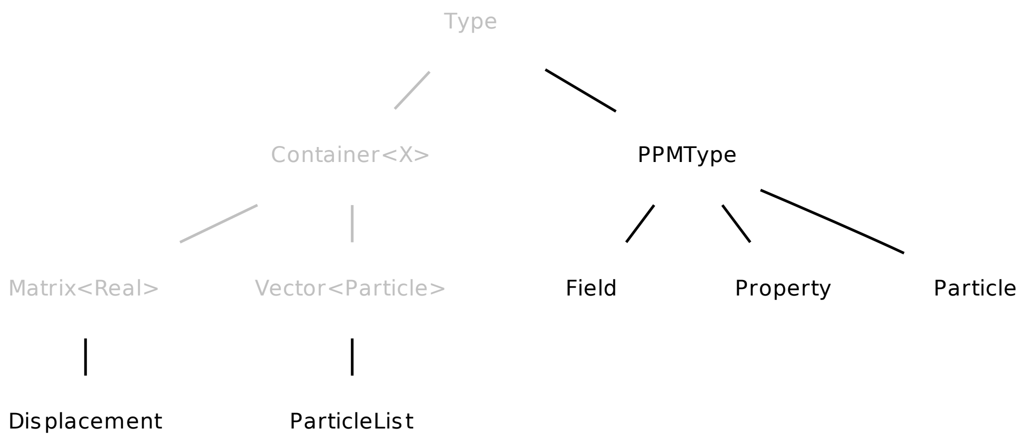


Figure 4.2.: The domain-specific extensions $\mathcal{T}_{PPM}$ of `de.ppme.core` to the base hierarchy.

4.1.2. Type Inference

PPME's language model is statically typed. This means that all type annotations are known at compile time. As a consequence, all variable declarations in a PPME client need to have their type explicitly designated by the developer.

In order to be able to speak about type-inference in PPME, we need to have a notion of the abstract syntax of our language. The syntax of PPME's language can be summarized as a collection of syntactic sets, each corresponding to a different kind of program phrase. Overall, these syntactic sets can be grouped into two logical parts. *Basic syntactic sets* are very simple and form the groundwork for PPME programs. These can be taken as given, and, in the typing system, elements from the basic set are trivially mapped to corresponding types. Leveraging this foundation, *derived syntactic sets* make up the more complex expressions of PPME clients. They allow the developer to build more complex expressions, e. g., unary and binary expressions, or domain-specific directives such as differential operators.

In the following, both sets are introduced and the related typing rules are explained.

Basic Syntactic Sets The basic syntactic sets in PPME are comprised mainly of *literals*, notations for representing fixed values, and *variables*. We focus on the literals for primitive types with the notation agreements as shown in Table 4.1.

booleans	b	$b \in \mathbb{B} = \{true, false\}$
strings	s	e. g., $s = $ "PPME"
integers	n, m	$n, m \in \mathbb{N}$
reals	r	e. g., $r = 3.14$ or $r = 6.62\text{E}{-34}$

Table 4.1.: Literals for the primitive types in PPME and their notational agreements.

The boolean truth values are ranged over by (the metavariable) b. We use s to range over strings, i. e., character sequences enclosed by quotes. n, m denote integers, and r usually refers to a real number. In particular, real numbers can be written in floating-point notation, e. g., $r = 3.14$, or in scientific notation, e. g., $r = 6.62\text{E}{-34}$ [IEE08].

The literals have type information associated in a natural way, as can be seen in the following equations. Obviously, truth values have type $Boolean$, integers have type $Integer$, and so on.

$$\Gamma \vdash b : Boolean \qquad \text{if } b \text{ is } \texttt{BooleanLiteral} \tag{4.1}$$

$$\Gamma \vdash n : Integer \qquad \text{if } n \text{ is } \texttt{IntegerLiteral} \tag{4.2}$$

$$\Gamma \vdash r : Real \qquad \text{if } r \text{ is } \texttt{DecimalLiteral} \text{ or } \texttt{ScientificNumber} \tag{4.3}$$

$$\Gamma \vdash s : String \qquad \text{if } s \text{ is } \texttt{StringLiteral} \tag{4.4}$$

Additionally, variables are a basic syntactic set. We typically denote variables by v or by using some identifier such as x_n.

variables	v	$v \in Var = \{a, b, \ldots, x, x', x_1, x_2, \ldots\}$

Table 4.2.: Variables and their notation.

The type of a variable depends on the typing environment Γ. If the environment contains an entry for a variable v, i. e., there is $\langle v, \tau \rangle \in \Gamma$, then the type of v is equal to τ. A lookup of v in Γ is denoted by $\Gamma(v)$. This leads to the first typing rule we specify. Equation (4.5) states that if v is in the environment and has type τ, then an occurrence of v is typed with τ as well.

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau} \tag{4.5}$$

Variables are added to the typing environment at the point where they are declared. Given a typing environment Γ, the declaration of a variable yields a new typing environment Γ' which contains an entry for the newly declared variable. Since MPS is based on a structural editor, identifiers for variables can be ambiguous (although that is not recommended as it will likely lead to conflicts in the generated code) as the developer can simply select the intended variable reference. The type system, however, is able to resolve ambiguous variable names. For variable declaration, with and without initialization, rule (4.6) applies.

$$\frac{}{\Gamma \vdash \tau\ x : \Gamma \cup \{x = \tau\}} \qquad \frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash \tau\ x = e : \Gamma \cup \{x = \tau\}} \tag{4.6}$$

Derived Syntactic Sets Besides the simple sets we have seen beforehand, more complex syntactic sets can be derived. The abstract syntax of derived syntactic sets is given by the possible forms of expressions e in PPME.

Unary Operations $\ominus(e)$

PPME supports three unary operations, the unary minus $-e$, the logical not $!e$, and the square root $\sqrt{e}$. Obviously, this definition allows for "pointless" expressions such as taking the square root of a string. The remainder of this section will present rules for well-formedness and type conclusion to prevent these erroneous phrases.

$$\ominus \in \{-, !, \sqrt{}\}$$

Binary Operations $e_1 \otimes e_2$

Binary operations come in various shapes. First, PPME allows for typical arithmetic expressions like *addition*, *subtraction*, *multiplication*, *division*, and *exponentiation*.

$$BinOp_{arith} = \{+, -, *, /, \hat{}\}$$

Secondly, we have two operators for logical *and* and *or*.

$$BinOp_{log} \quad = \{\&\&, ||\}$$

Thirdly, the common operators for (in-) equality and less/greater than (or equal to) comparison of two expression are available.

$$BinOp_{comp} = \{==, != , <, >, <=, >=\}$$

The union of these three categories yields the set of binary operations in PPME. As for unary operations, well-formedness of binary expressions will be considered, and typing rules will be applied to decide on the resulting type.

$$\otimes \in BinOp_{arith} \cup BinOp_{log} \cup BinOp_{comp}$$

Domain-specific Operations

A strength of PPME are its domain-specific operations. They extend the set of basic operations and integrate seamlessly into the language. To cite an example, they allow for concise notations of mathematical concepts, preserving the expressiveness of the mathematical notation.

In PPME, fields and properties are defined on particle lists. Therefore, the language offers means to access these fields and properties given a particle list. This syntactic concept is called a *particle list access* (PLA). Given a particle list ps, a field f and a property x both defined on ps, then

$$ps \to f, \qquad ps \to x$$

denote the access of field f and property x, respectively. Intuitively, the result of a PLA is the whole field or property over the particle list.

In an analogous manner, the value of a field or property on a specific particle can be accessed via a *particle access* operation (PA). Likewise, the access operation is denoted by an arrow. Given a particle member p of the particle list ps, the fields and properties of ps can be accessed for p:

$$p \to f, \qquad p \to x$$

Additionally, PPME allows the developer to access the implicit properties of a particle, e. g., its position:

$$p \to pos$$

Intuitively, the result of a PA operation is the value of the field or property on particle p.

Finally, there are domain-specific notations for differential operators. Simulation developers can use these operators in partial differential equations (PDEs), staying close to the mathematical notation. To cite an example, the Laplacian operator $\nabla^2 e$ is used in the Gray-Scott reaction-diffusion scenario presented in section 3.3.2.

Access Operations $v[i]$, $m[i][j]$

As PPME provides the developer with various container types (e. g., vectors and matrices) it also offers means to access elements of these array-like structures. Access operations are denoted by square brackets containing the index to access.

Similarly, elements of non-scalar particle fields and properties can be accessed using the same notation. Be ps a particle list with a non-scalar field f, and $p \in ps$ a particle from ps, then $p \to f[i]$ denotes the access of the i^{th} element of f.

Type Rules Given the syntactic sets presented above, in the following, the corresponding type rules will be examined. Based on the intuitive typing of the basic syntactic sets, the type of a derived expression e can be inferred by the kind of e and its subexpressions.

We begin with a simple rule for parenthesized expressions. Expressions in PPME can be parenthesized to specify (or clarify) the order of operations which does not change the type of the expression. The type-inference for a parenthesized expression is the trivial rule (4.7).

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e) : \tau} \tag{4.7}$$

As elaborated above, PPME provides two kinds of unary expressions, namely unary arithmetic expressions and unary logic operations. The unary arithmetic operations are only defined for expression of a numeric type, i. e., $\tau \in \{Integer, Real\}$. Whereas the unary minus operator does not change the type of the expression (4.8), the resulting type of the square root operation is always a $Real$ (4.9).

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{Integer, Real\}}{\Gamma \vdash -e : \tau} \tag{4.8}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{Integer, Real\}}{\Gamma \vdash \sqrt{e} : Real} \tag{4.9}$$

On the other hand, unary logic operations are only defined for expressions of type $Boolean$. The result of negating a boolean expression is again a truth value (4.10).

$$\frac{\Gamma \vdash e : Boolean}{\Gamma \vdash !e : Boolean} \tag{4.10}$$

The following explanations address binary operations in PPME. A binary expression is characterized by the operation and its two subexpressions.

We begin with the *assignment expression* which denotes the assignment of an expression e to a variable x. The premise for this expression is that the type τ' of the assigned expression e is compatible with the variable type τ, i. e., $\tau' \leq \tau$, τ' is a subtype of τ. The resulting type of the assignment is then the variable type τ (4.11).

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash e : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash x = e : \tau} \tag{4.11}$$

Going over the binary operations available in PPME, recall the three groups of operations: arithmetic, logical, and relational operations. We begin with the type rule for logical operations (4.12) as it is the most simple one. Both subexpressions e_1 and e_2 have to be of type $Boolean$, and the result of a logical operation is again a logical value.

$$\frac{\Gamma \vdash e_1 : Boolean \qquad \Gamma \vdash e_2 : Boolean}{\Gamma \vdash e_1 \otimes e_2 : Boolean} \quad \text{where } \otimes \in BinOp_{log} \tag{4.12}$$

Similarly, the result of a relational operation is a logical value as well. However, the left and right side of a relation can have arbitrary types as long as they are *comparable*. In order to infer the resulting type of relational operations, it has to be known which types are comparable. With a look at the types defined in PPME, full comparability is only given between the numeric types $Integer$ and $Real$. We denote comparable types by $\tau_1 \perp \tau_2$, meaning that instances of the types τ_1 and τ_2 are comparable.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \otimes e_2 : Boolean} \quad \text{where } \otimes \in BinOp_{comp}, \tau_1 \perp \tau_2 \tag{4.13}$$

For binary arithmetic operations the general type rule (4.14) can be given. It states that the type of the operation $\otimes$ is dependent on the types of the subexpression e_1 and e_2 and the operation itself, i. e., it is *overloaded* for different operand types. We use $\tau_\otimes(\tau_1, \tau_2)$ to denote the type resolution for the given operation and subexpression types.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \otimes e_2 : \tau} \qquad \text{where } \otimes \in BinOp_{arith}, \tau = \tau_\otimes(\tau_1, \tau_2) \tag{4.14}$$

This type resolution for a specific operator $\otimes$ can be notated in a table. Tables 4.3 to 4.6 show the inference tables for the arithmetic operations in PPME. In the tables, abbreviated forms for the types are used. Additionally, we use $\uparrow(\tau_1, \tau_2)$ to denote the least common supertype of τ_1 and τ_2. Note that the type-inference for the element types of vectors, fields, and properties is deferred to another type check $\tau_\otimes$. This design decision reflects the mathematical nature of the operation, e. g., when multiplying a scalar value with a vector. Moreover, note that not all combinations of operand types yield a resulting type. The consequences of such undefined operation types are discussed in Section 4.1.3.

As elaborated above, PPME provides access operations for the container types. The operations always depend on the component type of the accessed structure and require an expression of type $Integer$ as positional index. Therefore, the type rules for the container types look similar to each other. Equations (4.15) and (4.16) show the rules for access operations on the base container types vector and matrix. Accessing a one-dimensional vector requires one index expression n, whereas accessing a two-dimensional matrix requires two index expressions n and m. To further improve the handling of vectors and matrices, bound checks for index expressions can be added to PPME.

$$\frac{\Gamma \vdash v : Vector\langle\tau\rangle \qquad \Gamma \vdash n : Integer; n \geq 0}{\Gamma \vdash v[n] : \tau} \tag{4.15}$$

$$\frac{\Gamma \vdash v : Matrix\langle\tau\rangle \qquad \Gamma \vdash n, m : Integer; n, m \geq 0}{\Gamma \vdash m[n][m] : \tau} \tag{4.16}$$

Similarly, particle-access operations are resolved. Recall that particle access is denoted by the arrow operator $\rightarrow$, and thus $p{\rightarrow}f$ means the access of f on p. For scalar fields and properties, the return type is equal to the type of the field or properties. In case that the field or property is non-scalar, the return type is a vector derived from the original structure. Equations (4.17) and (4.18) show the rules for accessing a field or property $\mathcal{E}\langle\tau, n\rangle$ with type τ and dimension n defined on the particle p.

$$\frac{\Gamma \vdash p : Particle \qquad \Gamma \vdash f : \mathcal{E}\langle\tau, 1\rangle}{\Gamma \vdash p \rightarrow f : \tau} \tag{4.17}$$

$$\frac{\Gamma \vdash p : Particle \qquad \Gamma \vdash f : \mathcal{E}\langle\tau, n\rangle, n \geq 2}{\Gamma \vdash p \rightarrow f : Vector\langle\tau\rangle} \tag{4.18}$$

(where $p \in ps : ParticleList$, $\mathcal{E}$ is a field or property defined on ps)

In consequence, access to elements of non-scalar fields and properties (where dimension $n \geq 2$) is simply an access to a vector. The following rule illustrates this inference.

$$\frac{\Gamma \vdash p \rightarrow f : \mathcal{E}\langle\tau, n\rangle \qquad n \geq 2 \qquad \Gamma \vdash i : Integer}{\Gamma \vdash (p \rightarrow f)[i] : \tau}$$

$\tau_+(\tau_1,\tau_2)$ $\tau_-(\tau_1,\tau_2)$	I	R	$V\langle X\rangle$	$F\langle X,n\rangle$	$P\langle X,n\rangle$
I	I	R	$V\langle\uparrow(I,X)\rangle$	$F\langle\uparrow(I,X),n\rangle$	$P\langle\uparrow(I,X),n\rangle$
R	R	R	$V\langle\uparrow(R,X)\rangle$	$F\langle\uparrow(R,X),n\rangle$	$P\langle\uparrow(R,X),n\rangle$
$V\langle Y\rangle$	$V\langle\uparrow(Y,I)\rangle$	$V\langle\uparrow(Y,R)\rangle$	$V\langle\uparrow(Y,X)\rangle$	—	—
$F\langle Y,m\rangle$	$F\langle\uparrow(Y,I),m\rangle$	$F\langle\uparrow(Y,R),m\rangle$	—	$F\langle\uparrow(Y,X),n\rangle^\dagger$	—
$P\langle Y,m\rangle$	$P\langle\uparrow(Y,I),m\rangle$	$P\langle\uparrow(Y,R),m\rangle$	—	—	$P\langle\uparrow(Y,X),n\rangle^\dagger$

† if $n=m$

$I=Integer$, $R=Real$, $V\langle X\rangle=Vector\langle X\rangle$, $F\langle X,n\rangle=Field\langle X,n\rangle$, $P\langle X,n\rangle=Property\langle X,n\rangle$.

Table 4.3.: Type inference table for the binary multiplication operators $+$ and $-$.

$\tau_*(\tau_1,\tau_2)$	I	R	$V\langle X\rangle$	$F\langle X,n\rangle$	$P\langle X,n\rangle$
I	I	R	$V\langle\uparrow(I,X)\rangle$	$F\langle\uparrow(I,X),n\rangle$	$P\langle\uparrow(I,X),n\rangle$
R	R	R	$V\langle\uparrow(R,X)\rangle$	$F\langle\uparrow(R,X),n\rangle$	$P\langle\uparrow(R,X),n\rangle$
$V\langle Y\rangle$	$V\langle\uparrow(Y,I)\rangle$	$V\langle\uparrow(Y,R)\rangle$	—	—	—
$F\langle Y,m\rangle$	$F\langle\uparrow(Y,I),m\rangle$	$F\langle\uparrow(Y,R),m\rangle$	—	—	—
$P\langle Y,m\rangle$	$P\langle\uparrow(Y,I),m\rangle$	$P\langle\uparrow(Y,R),m\rangle$	—	—	—

$I=Integer$, $R=Real$, $V\langle X\rangle=Vector\langle X\rangle$, $F\langle X,n\rangle=Field\langle X,n\rangle$, $P\langle X,n\rangle=Property\langle X,n\rangle$.

Table 4.4.: Type inference table for the binary multiplication operator $*$.

$\tau_/(\tau_1,\tau_2)$	I	R	$V\langle X\rangle$	$F\langle X,n\rangle$	$P\langle X,n\rangle$
I	R	R	$V\langle\tau_/(I,X)\rangle$	$F\langle\tau_/(I,X),n\rangle$	$P\langle\tau_/(I,X),n\rangle$
R	R	R	$V\langle\tau_/(R,X)\rangle$	$F\langle\tau_/(R,X),n\rangle$	$P\langle\tau_/(R,X),n\rangle$
$V\langle Y\rangle$	$V\langle\tau_/(Y,R)\rangle$	$V\langle\tau_/(Y,R)\rangle$	—	—	—
$F\langle Y,m\rangle$	$F\langle\tau_/(Y,R),m\rangle$	$F\langle\tau_/(Y,R),m\rangle$	—	—	—
$P\langle Y,m\rangle$	$P\langle\tau_/(Y,R),m\rangle$	$P\langle\tau_/(Y,R),m\rangle$	—	—	—

$I=Integer$, $R=Real$, $V\langle X\rangle=Vector\langle X\rangle$, $F\langle X,n\rangle=Field\langle X,n\rangle$, $P\langle X,n\rangle=Property\langle X,n\rangle$.

Table 4.5.: Type inference table for the binary division operator $/$.

$\tau_{a^b}(\tau_1,\tau_2)$	I	R	$V\langle X\rangle$	$F\langle X,n\rangle$	$P\langle X,n\rangle$
I	I	R	—	—	—
R	R	R	—	—	—
$V\langle Y\rangle$	$V\langle\tau_{a^b}(Y,I)\rangle$	$V\langle\tau_{a^b}(Y,R)\rangle$	—	—	—
$F\langle Y,m\rangle$	$F\langle\tau_{a^b}(Y,R),m\rangle$	$F\langle\tau_{a^b}(Y,R),m\rangle$	—	—	—
$P\langle Y,m\rangle$	$P\langle\tau_{a^b}(Y,R),m\rangle$	$P\langle\tau_{a^b}(Y,R),m\rangle$	—	—	—

$I=Integer$, $R=Real$, $V\langle X\rangle=Vector\langle X\rangle$, $F\langle X,n\rangle=Field\langle X,n\rangle$, $P\langle X,n\rangle=Property\langle X,n\rangle$.

Table 4.6.: Type inference table for the exponentiation operation a^b.

In mathematics, a differential operator can be seen as an abstract operation that takes a function and returns another function. In PPME, we focus on discretized differential operators, so that they have a meaning on a discretized grid. Furthermore, the occurrences of differential operators in PPME are restricted to take fields with numerical type as operand. Under this assumption, the following type rule (4.19) is used to model the behavior of differential operators such as the Laplacian. The general output type for differential operators is real.

$$\frac{\Gamma \vdash D : DiffOp \qquad \Gamma \vdash e : Field\langle \tau, n \rangle}{\Gamma \vdash D(e) : Field\langle Real, n \rangle} \tag{4.19}$$

Equations (4.5) to (4.19) specify the type rules for valid and well-formed expressions. Given these rules and their implementation in PPME, MPS is able to infer and check types in a simulation program. Section 4.1.3 elaborates what happens to non-well-formed, erroneous expressions and how they are communicated to the developer. Finally, Section 4.1.4 provides more information on how the previously defined type rules are implemented in PPME using they type system language aspect of MPS.

4.1.3. Typing Errors

As aforementioned, all expressions matching one the rules presented above is *well-formed* and can be typed by the system. However, it might occur that a user enters a faulty expression for which a type cannot be inferred, yielding a *typing error*. In this case, PPME has to communicate the error to the developer and provide meaningful information on where the error is located. In the following, the process of identifying typing errors and reporting them to the developer is discussed.

A common approach for catching typing errors is to introduce an *error type* as the result of non-well-formed expressions [Plo81; Plo04]. For this purpose PPME uses the `RuntimeErrorType` provided by MPS which can hold further information on the cause and location of the error. The error occurs at the editor's runtime, but marks a static error in the simulation code. It contains a references to the node which caused the error, and an additional error message which is presented to the developer to guide him or her to resolve the issue.

In the editor, typing errors are indicated by red error marks, similar the ones known from classical syntax checks. Hovering the mouse cursor over the erroneous node will show the error message in a pop-up window, helping the developer correcting the expression. In case MPS is not able to infer the type of an expression, e. g., because the type of one or more subexpressions is never concrete and thus cannot be determined, it indicates a type warning using the common orange warning marker. Note, however, that in most cases this indicates a problem with the type system implementation rather then with the code.

There are different causes for typing errors. The first occurs when types are incompatible to each other, e. g., when checking the subtyping rule in an assignment expression (cf. Rule (4.11)) or the comparability of two types for a relational expression (cf. Rule (4.13)). Generally speaking, it all falls back on operators not being defined for specific operand types. To cite an example, the exponentiation of a scalar with a vector is not a meaningful mathematical operation and yields to typing error in the exponentiation expression. Finally, errors might be propagated to invalidate the parenting expression. Remember the resolution table 4.3 for addition and subtraction. Given two vectors, $v_1 : Vector\langle Real \rangle$ and $v_2 : Vector\langle Boolean \rangle$, the addition of the two vectors $v_1 + v_2$

depends on the resolution of $\tau_+(Real, Boolean)$ which is not defined. This error is propagated back up to the vector addition and reported to the user.

These observations lead to the following *error rules* for unary and binary operations. We have generalized the type resolution for unary operations $\ominus$ in order to use a similar notation. A typing error is indicated by $\mathbb{E}$. For unary operations $\ominus$ the typing depends on the operation and its operand. $\tau_\ominus$ yields an error $\mathbb{E}$ in the cases that are not defined in the typing rules, e. g., when using a unary arithmetic operation on truth values.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \ominus(e) : \mathbb{E}} \qquad \text{if } \tau_\ominus(\tau) = \mathbb{E} \tag{4.20}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \otimes e_2 : \mathbb{E}} \qquad \text{if } \tau_\otimes(\tau_1, \tau_2) = \mathbb{E} \tag{4.21}$$

With these remarks on typing errors we conclude the formal part on type rules in PPME. Section 4.1.4 addresses their implementation using the language aspects provided by MPS. For some of the presented type rules, the corresponding implementation is exemplary discussed.

4.1.4. Implementation in PPME

As pointed out before, any language element can function as a type in MPS. We have already seen in Section 4.1.1 how types itself are implemented in PPME, building a hierarchy of types. Now we put emphasis on the implementation details for type-inference rules and the sub-/supertype mechanisms. As the MPS documentation states, a type system is a part of the language definition which assigns types to the nodes within the model [MPS15c]. The integrated *type system language* allows to check certain constraints or conditions tied to nodes and their types. In particular, the type rules elaborated above are implemented using the type system language and its concepts of *inference rules* (`typeof_` rules) and *overloaded binary operators* (`overload_` rules).

We take the type-inference rule (4.14) for binary arithmetic operations as an example and present the corresponding implementation Listing 4.1. The second line states that the rule is generally applicable to all binary expressions. Lines 6 to 10 declare the dependencies on the types of left and right operand. The `operation type` directive (l. 12) matches the type resolution $\tau_\otimes(\tau_{left}, \tau_{right})$, where $\otimes$ is given by the operation `be`, and $\tau_{left}, \tau_{right}$ correspond to `leftType`, `rightType`, respectively. In case the operation type cannot be determined, an error message is generated in accordance with section 4.1.3 (l. 16). Otherwise, the type of the binary expression `be` is equal to (`:==:`) the operation type `opType`.

The file `overload_BinaryArithmeticOperation` in the implementation package `de.ppme.expressions` models the type resolution for base types $\mathcal{T}_{base}$ and base operations, i. e., the arithmetic operations of addition, subtraction, multiplication, and division. The resolution table is completed by overloading rules for domain-specific types, defined in the `de.ppme.core` package.

The first rule (ll. 3–11) shown in Listing 4.2 is applicable to addition, subtraction, and multiplication (line 3). It delegates the type-inference to a utility method (line 10) which basically determines the least common ancestor for the two given types in the type hierarchy.

The second rule (ll. 13–21) states that dividing two primitive types yields a result of type $Real$, as denoted in Table 4.5.

The third rule (ll. 23–39) describes the type resolution for vector-scalar multiplication and division, i. e., one operand has to be of vector type (l. 24) and the other operand has

```
rule typeof_BinaryExpression {
  applicable for concept = BinaryExpression as be
  overrides false

  do {
    addDependency(be.left);
    addDependency(be.right);

    when concrete (typeof(be.left) as leftType) {
      when concrete (typeof(be.right) as rightType) {
        node<> opType = operation type(be, leftType, rightType);
        if (opType.isNotNull && !opType.isInstanceOf(RuntimeErrorType)) {
          typeof(be) :==: opType;
        } else {
          error "Operator '" + be + "' cannot be applied to '" + leftType + "', '" +
              ↪ rightType + "'" -> be;
        }
      }
    }
  }
}
```

Listing 4.1: Type inference rule for binary operations.

to be of a plain number type `INumber` (l. 27). In lines 32 and 33 the involved types are extracted from the operation expression for further processing. Line 34 delegates the type resolution to the application of the operation to the vector's elements to determine the component type of the resulting vector. Again, in case the operation type cannot be inferred by the system, the user is alerted by an error message. Otherwise, the resulting vector type is constructed and returned (l. 38).

The last rule shown (ll. 41–57) defines the type for vector-vector addition and subtraction. As with the rule before, the inference of the component type is delegated (l. 51) and in case of success the resulting vector type is constructed and returned.

The remaining rules for domain-specific types look very similar and follow the resolutions defined in Tables 4.3 to 4.6. Being able to translate the formalized type rules directly into the type system language of MPS allows for a good understanding of the implementation and provide a flexible scope for type system extensions and refinements.

As mentioned before, another important part of the MPS type system are super- and subtype relations. When the type system engine solves inequations over types, it requires information about whether a type is a subtype of another type [MPS15c]. This is information is given by *subtyping rules*. One advantage of using MPS is that the type system engine is able to resolve supertypes automatically. When checking whether some type A is a supertype of another type B, it applies subtyping rules to B and computes its immediate supertypes, then applies subtyping rules to those supertypes and so on. If type A is among the computed supertypes of type B, the check is successful [MPS15c].

```
subtyping rule subtype_integer_real {
  weak = false
  applicable for concept = IntegerType as integerType

  supertypes {
    return <RealType()>;
  }
}
```

Listing 4.3: Subtyping rule for the immediate supertypes of $Integer$.

```
overloaded operations rules overload_BinaryArithmeticOperation

operation concepts: PlusExpression | MinusExpression | MulExpression
left operand type: <PrimitiveType()> is exact: false use strong subtyping true
right operand type: <PrimitiveType()> is exact: false use strong subtyping true
is applicable:
  <no isApplicable>
operation type:
(operation, leftOperandType, rightOperandType)->node<>
  Queries.getBinaryOperationType(leftOperandType, rightOperandType);
}
-----------------------------------------------
operation concepts: DivExpression
left operand type: <PrimitiveType()> is exact: false use strong subtyping true
right operand type: <PrimitiveType()> is exact: false use strong subtyping true
is applicable:
  <no isApplicable>
operation type:
(operation, leftOperandType, rightOperandType)->node<>
  Queries.getBinaryOperationType(Queries.getBinaryOperationType(leftOperandType, rightOperandType), <RealType()>);
}
-----------------------------------------------
operation concepts: MulExpression | DivExpression
one operand type: <AbstractVectorType()> is exact: false use strong subtyping false
is applicable:
(leftOperandType, rightOperandType, operation)->boolean
  leftOperandType.isInstanceOf(VectorType) && rightOperandType.isInstanceOf(INumber) || rightOperandType.
        ↪ isInstanceOf(VectorType) && leftOperandType.isInstanceOf(INumber);
}
operation type:
(operation, leftOperandType, rightOperandType)->node<>
  node<VectorType> vectorType = leftOperandType.isInstanceOf(VectorType) ? leftOperandType as VectorType :
        ↪ rightOperandType as VectorType;
  node<INumber> numberType = leftOperandType.isInstanceOf(VectorType) ? rightOperandType as INumber :
        ↪ leftOperandType as INumber;

  node<Type> dtype = operation type(operation, vectorType.componentType, numberType) as Type;
  if (dtype.isNull) {
    return <RuntimeErrorType(errorText: "Operation cannot be applied to operands (could not determine vector
          ↪ component type) - vector type: " + vectorType.componentType + ", number type: " + numberType)>;
  }
  return <VectorType(componentType: # dtype)>;
}
-----------------------------------------------
operation concepts: PlusExpression | MinusExpression
left operand type: <AbstractVectorType()> is exact: false use strong subtyping false
right operand type: <AbstractVectorType()> is exact: false use strong subtyping false
is applicable:
<no isApplicable>
operation type:
(operation, leftOperandType, rightOperandType)->node<>
  node<VectorType> leftType = leftOperandType as VectorType;
  node<VectorType> rightType = rightOperandType as VectorType;

  node<Type> dtype = operation type(operation, leftType.componentType, rightType.componentType) as Type;

  if (dtype.isNull) {
    return <RuntimeErrorType(errorText: "Operation cannot be applied to operands (could not determine vector
          ↪ component type) - left: " + leftType + ", right: " + rightType)>;
  }
  return <VectorType(componentType: # dtype)>;
}
```

Listing 4.2: Type resolution rules for overloaded binary expressions in `de.ppme` ↪ `.expressions`.

Listing 4.3 shows a very simple example, the subtyping rule for $Integer$, denoting that it has $Real$ as immediate supertype. The subtyping is *strong*, i. e., not weak (l. 2), stating that an integer can always be used instead of a real. This models the fact that an integer is also a real number. Line 3 states that this rule can be applied to concepts of type `IntegerType`. The `supertypes` block (ll. 5–7) is responsible for building the list of immediate supertypes. In this case, the only direct supertype `RealType` is returned.

The rule shown in Listing 4.4 is an example for a more complex rule regarding the type system engine. Instead of a subtyping rule, a *replacement rule* is used. Replacement rules help to solve type-system inequations by removing the inequation in case the rule is applicable and executing the rule's body. Usually, the body of a replacement rule will add new equations and inequations [MPS15c]. The specific rule shown models the covariant subtyping relation of the vector type, i. e., if $A \leq B$ for two types A and B then $Vector\langle A\rangle \leq Vector\langle B\rangle$ (see lines 9 and 15). Similar to the subtyping rule shown Listing 4.3 a replacement rule is only applicable to specific elements. This restriction is denoted by line 3, limiting the rule's scope to type system equations for two vector types, named `left` and `right`.

```
1  replacement rule vectorType_subtypeOf_VectorType
2
3  applicable for  concept = VectorType as left <: concept = VectorType as right
4
5  custom condition: true
6
7  rule {
8    if (left.ndim.isNull) {
9      infer left.componentType :<=: right.componentType;
10   } else if (left.ndim.isNotNull && right.ndim.isNotNull) {
11     int ldim = left.ndim.isCompileTimeConstant() ? left.ndim.getCompileTimeConstant() as Integer : -1;
12     int rdim = right.ndim.isCompileTimeConstant() ? right.ndim.getCompileTimeConstant() as Integer : -1;
13
14     if (ldim == rdim) {
15       infer left.componentType :<=: right.componentType;
16     }
17   }
18 }
```

Listing 4.4: Replacement rule for capturing covariant subtyping on vector types.

Moreover, the rule is already geared to cover size specifications for vectors, an improvement recently added to PPME. For reasons of compatibility, the size of the right type is ignored if the size property of the left type is not defined. One could say the size information is erased in this case. If the dimension specification `ndim` of both vectors is not empty and can be evaluated statically, the vector sizes are compared (ll. 10–16).

4.2. Physical Unit Annotations

Adding a notion of measurement to a programming language benefits software development in many ways. Especially in physical science this effect is noticeable. Verifying dimensional integrity prevents erroneous expressions and equations which are hard to detect otherwise. Such an additional level of analysis to detect inconsistencies improves the quality of programs in a substantial way. Moreover, units of measurement annotated to variables and expressions are good documentation for the program itself. Often, units are either comments in the code or not endorsed at all, which in both cases means that they are not used to verify the program with all information available. Therefore, we present a language extension to PPME to support physical-unit annotations in particle-mesh simulations.

The idea of incorporating physical units of measurement into computer programs is not new but dates back more than five decades [Che60]. During this period of time many approaches to this subject were taken. Early work by Karr and Loveman presented a “units calculus”, a method to manage relationships and conversions of units, to incorporate in programming languages [KL78]. House further develops the ideas of Gehani of extending an existing language with new type constructs for units, addressing the concern of static analysis [Hou83; Geh77]. Cmelik and Gehani and Umrigar extended the idea of units of measure to general *dimensional analysis*. Dimensional analysis covers logical classes of units, e. g., distance or mass, meaning that quantities with the same dimension but different units differ only in a factor. Their work covers dimensional analysis with C++ at both runtime and compile-time, making extensive use of template metaprogramming [CG88; Umr94]. Simultaneously, physical-unit extensions to type systems in functional languages were developed by various authors. Dimensional analysis fits neatly into the concept of type inference in in these languages, establishing a base for units and dimensions in functional languages [WO91; Ken94; Ken97; HM95]. More recently, Cook et al. presented an analysis technique checking correctness of units in program without extending the base language, aiming for a minimal effort of annotations for a developer [CFH06]. Austin proposed unit annotations for matrix and finite element calculations [Aus06].

However, none of these approaches has stuck out to be a near-optimal solution. Often, the integration of units into a programming language has flaws. For instance, framework approaches in object-oriented languages often use abstractions with boxing and unboxing of quantities and units, implying a runtime overhead for analysis. Voelter cites some examples for drawbacks of other conventional implementations in the context of C programs for embedded systems [Voe14]. One approach integrates unit information in form of comments in the source code which are then processed by external tools to verify integrity. Obviously, this approach suffers from bad syntactic integration. Moreover, the external verification tool does not only need a parser for the unit annotations but also for the C program itself. In contrast to that, macros can be used to incorporate units of measure for types and values with regard to available units. However, still external tools are required for compile-time unit checking. In conclusion, Voelter sees the solution in real language extension of a base language with means to annotate types and values with unit information.

The strength of PPME and MPS as underlying workbench technology lies in the modularity and extensibility of languages. This implies that physical units annotations and dimensional analysis can be implemented as an (optional) language extension for the simulation language in PPME. The core language is independent of the extension, meaning that a developer can chose whether to use units in a simulation or not, and that existing programs can be annotated with unit information subsequently.

In the following, we present a model for unit annotations and the `PhysUnits` language extension for PPME. In Section 4.2.1, the concepts of elementary and derived units, and unit specifications are introduced. Further, we elucidate how the type inference rules in Section 4.1 are adapted to cover unit annotations. Finally, Section 4.2.2 covers the technical implementation of the proposed model in PPME.

4.2.1. Units in PPME

Units of measure are an optional feature in PPME which are logically annotated to types and expression and processed by the type system. They are an additional layer supplementing the existing type environment with extra information. In the following, firstly unit declarations for elementary and derived units are presented. These allow developers to freely define the units and relations among them. Secondly, we show how unit annotations influence the type system. This includes adaption and overloading of type rules as well as mechanics for unit inference.

Unit Declarations The incorporation of units follows loosely the suggestions by Karr and Loveman in [KL78]. The developer is free to define the elementary units required for the specific use case, and derived units for the sake of readability. These declarations are independent of the language itself, but can be reused among different simulation programs.

Unit declarations in PPME reside in a special file owned by a model. Each declaration describes a unit with an identifier `<u>`, an optional specification `<spec>`, and an optional suggestive name `<desc>`.

```
<u>  := <spec> (<desc>)
```

A unit declaration without specification is called an *elementary unit* or *base unit* as it cannot be decomposed. We denote the set of all elementary units by $\mathcal{E}$. The following listing shows two examples for the definition of base units for meter and second, respectively. Often, the base units used in simulations correspond to SI units commonly used in physical sciences, but other units are possible, e. g., currencies or abstract units of measure.

```
m   := <no spec> (meter)
s   := <no spec> (second)
```

The specification field is used to define units which are derived from other units, so called *derived units*. For instance, a unit to measure velocity, meters per second (`mps`), is expressed in terms of the two elementary units defined above ($mps = \frac{m}{s} = m \cdot s^{-1}$).

```
mps := m · s^-1 (meter per second)
```

Note that derived units in PPME can be composed from both elementary units and other derived units by means of multiplication and exponentiation. To cite an example, given another base unit `kg` for mass, we can derive Newton `N`, the unit for force, from the base units `kg`, `m`, and `s`. Joule `J`, a derived unit for energy, can then be expressed in terms of `N` and `m`.

```
kg := <no spec> (kilogram)
N  := kg · m · s^-1 (Newton)
J  := N · m (Joule)
```

Unit Specifications The unit calculus in PPME works with the units declared by the user. In order to treat physical unit annotations uniformly over elementary, derived, and computed units we introduce the notion of unit specifications. A *unit specification* $u \in \mathcal{U}$ has the form

$$u ::= d \mid u_1 \cdot u_2 \mid u^n$$

where d is a declared unit, u_1, u_2, and u are unit specifications, and $n \in \mathbb{Z}$. As mentioned above, developers can specify their own base and derived units and are not limited to some predefined set.

In order to perform computations and comparison on units, a uniform representation of unit specifications is required. Therefore, we introduce the notion of a specification in base form. A unit specification is in *base form* if it is a set of base units raised to some integer exponent where each base unit $b_i \in \mathcal{E}$ occurs at most once.

$$u := \{b_1^{n_1}, b_2^{n_2}, \dots, b_k^{n_k}\}$$

By construction, there exists a unique base form for each unit specification u. We say that u is *expanded*, denoted by $\lceil u \rceil$, if all derived units in u are recursively replaced by their specification, yielding the base form of u. Two unit specifications u_1 and u_2 are equivalent, denoted by $u_1 \equiv u_2$, if $\lceil u_1 \rceil = \lceil u_2 \rceil$. We say that the units u_1 and u_2 *match*.

Unit Inference Rules Physical unit annotations are logically attached to types and expressions. We utilize the type system engine for unit checking and unit inference, similar to the base type system. Thus, the new type and unit inference rules use the same notation as in Section 4.1. The unit calculus is only responsible for computation and inference of units. The type inference is usually deferred to the original type system.

Given a type τ and a unit specification u, we denote the *annotated type* by

$$\hat{\tau} = [\tau; u]$$

We say that τ is the primitive type of the annotated type $\hat{\tau}$. An expression e with primitive type τ and unit specification u is then denoted using $e : [\tau; u]$.

The annotation of unit information to types or literal expressions is denoted by braces.

$$\tau\{u\} : [\tau; u] \qquad e : \tau \mapsto e\{u\} : [\tau; u]$$

This allows for variable initializations such as `integer{s} t = 10{s}`, where the original declaration of an integer variable is lifted to have unit information attached.

Additionally, we use an analogous notation to type inference for *unit inference* for binary operations, denoted by $u = \mathcal{U}_{\otimes}(u_1, u_2)$, meaning that the resulting unit u is computed from the operands' units u_1 and u_2 and the operation $\otimes$.

Subtyping rules for annotated types are quite simple. We consider an annotated type $\hat{\tau}_1 = [\tau_1; u_1]$ to be a subtype of another annotated type $\hat{\tau}_2 = [\tau_2; u_2]$, denoted by $\hat{\tau}_1 \leq \hat{\tau}_2$, if $u_1 \equiv u_2$ and $\tau_1 \leq \tau_2$, i. e., if the units match and the primitive types fulfill the subtype relation. Moreover, each annotated type is a subtype of its primitive type, $[\tau; u] \leq \tau$.

The type inference rules of section 4.1.2 have to be adapted to deal with unit specifications. In the following, the necessary extensions to the type system are presented. We start with the declaration of variables with attached unit information. As mentioned before, types can be annotated with unit information to form an annotated type. A variable declaration with initialization requires to the annotated units u and u' of the variable type and the assigned expression to be equivalent. The type check, $\tau' \leq \tau$, is left to the type system as before. If the declaration is valid (or the variable is declared without initialization), the pair $\langle x, \hat{\tau} \rangle$ is added to the typing environment.

$$\frac{\Gamma \vdash e : [\tau'; u'] \qquad \tau' \leq \tau \qquad u' \equiv u}{\Gamma \vdash \tau\{u\}\ x = e : \Gamma \cup \{x = [\tau; u]\}} \tag{4.22}$$

Similarly, the unit specifications have to match in case of a variable assignment. However, PPME allows to assign unit-annotated expressions to unitless variables, erasing the unit specification (4.24). This keeps the physical unit annotation backwards compatible and allows developers to add units to existing programs iteratively.

$$\frac{\Gamma \vdash x : [\tau; u] \qquad \Gamma \vdash e : [\tau'; u'] \qquad \tau' \leq \tau \qquad u' \equiv u}{\Gamma \vdash x = e : [\tau; u]} \tag{4.23}$$

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash e : [\tau', u] \qquad \tau' \leq \tau}{\Gamma \vdash x = e : \tau} \tag{4.24}$$

Binary operations need the most attention. The unit calculus needs to integrate with the type system of the base language and provide additional checks and inference rules for units. For instance, addition and subtraction are only applicable to operands with matching unit specifications, as depicted in (4.25). The resulting type is defined by the type resolution for $\tau_{\otimes}(\tau_1, \tau_2)$, and if the units match simply the left operand's unit is taken.

$$\frac{\Gamma \vdash e_1 : [\tau_1, u_1] \qquad \Gamma \vdash e_2 : [\tau_2, u_2] \qquad u_1 \equiv u_2}{\Gamma \vdash e_1 \otimes e_2 : [\tau, u_1]} \tag{4.25}$$

where $\otimes \in \{+, -\},\ \tau = \tau_{\otimes}(\tau_1, \tau_2),$

The restriction of matching units is dropped for multiplication and division. If the operation is valid for the operand types, i. e., $\tau_{\otimes}(\tau_1, \tau_2) \neq \mathbb{E}$, the resulting unit is calculated for the specific operation. That implies $u = u_1 \cdot u_2$ for multiplication and $u = \frac{u_1}{u_2}$ for division.

$$\frac{\Gamma \vdash e_1 : [\tau_1, u_1] \qquad \Gamma \vdash e_2 : [\tau_2, u_2]}{\Gamma \vdash e_1 \otimes e_2 : [\tau, u]} \tag{4.26}$$
$$\text{where } \otimes \in \{*, /\},\ \tau = \tau_{\otimes}(\tau_1, \tau_2),\ u = \mathcal{U}_{\otimes}(u_1, u_2)$$

Additionally, multiplication and division by scalars is allowed. Again, the resulting type is defined through $\tau_{\otimes}(\tau_1, \tau_2)$. The unit specification does not change except for the second case in (4.28) where the unit-attached operand is the divisor. In this case, the resulting unit specification is the reciprocal of the unit u_2.

$$\frac{\Gamma \vdash e_1 : [\tau_1; u_1] \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \otimes e_2 : [\tau, u_1]} \tag{4.27}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : [\tau_2, u_2]}{\Gamma \vdash e_1 \otimes e_2 : [\tau, u]} \tag{4.28}$$
$$\text{where } \tau = \tau_{\otimes}(\tau_1, \tau_2),\ u = \begin{cases} u_2 & \text{if } \otimes = * \\ {u_2}^{-1} & \text{if } \otimes = / \end{cases}$$

Binary operations, where the operand types involve container types such as vectors, need special treatment. The container will usually have the unit specification annotated at the component type, the container type itself being unitless. The physical unit engine has to propagate the units inside the container for inference of the resulting unit specification. The determination of the resulting container type with annotated component type obeys similar rules as presented above. The outer type is defined by the type rule deferred to the type system, and the unit information is propagated into the container type. Equations (4.29) and (4.30) show the inference rules for the vector type being the left operand. Analogous to the rules above, the resulting unit is either taken from one operand (4.29) or is calculated from the specifications and the operation (4.30).

$$\frac{\Gamma \vdash e_1 : Vector\langle \tau_1 \rangle \qquad \Gamma \vdash e_2 : [\tau_2, u_2]}{\Gamma \vdash e_1 \otimes e_2 : Vector\langle [\tau; u_2] \rangle} \tag{4.29}$$
$$\text{where } \tau_{\otimes}(Vector\langle \tau_1 \rangle, \tau_2) = Vector\langle \tau \rangle$$

$$\frac{\Gamma \vdash e_1 : Vector\langle [\tau_1; u_1] \rangle \qquad \Gamma \vdash e_2 : [\tau_2, u_2]}{\Gamma \vdash e_1 \otimes e_2 : Vector\langle [\tau; u] \rangle} \tag{4.30}$$
$$\text{where } \tau_{\otimes}(Vector\langle \tau_1 \rangle, \tau_2) = Vector\langle \tau \rangle,\ \mathcal{U}_{\otimes}(u_1, u_2) = u$$

Exponentiation for a unit-attached expression is only defined for integer exponents in PPME. The exponent's value has to be known at compile-time in order to compute the resulting unit specification. The necessity of the exponent being constant at compile time stems from design of unit specifications which do not allow for symbolic computations which would render checks on units undecidable in many cases. However, the limitation to integer exponents is explained by the current implementation, which only allows for integer exponents on units. This issue is further discussed in Section 4.2.3.

$$\frac{\Gamma \vdash e : [\tau, u] \qquad \Gamma \vdash n : Integer}{\Gamma \vdash e^n : [\tau, u^n]} \tag{4.31}$$

Unit Errors Obviously, the addition of physical units bears an additional source of errors. Analogous to the type system errors presented in Section 4.1.3, the unit calculus produces error messages for the developer if it finds inconsistencies or cannot resolve units. For instance, addition and subtraction are only allowed if both operands have unit attachments and their units are equivalent. Thus, we can derive the following exemplary error rules for the unit check:

$$\frac{\Gamma \vdash e_1 : [\tau_1, u_1] \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 + e_2 : \mathbb{E}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : [\tau_2, u_2]}{\Gamma \vdash e_1 + e_2 : \mathbb{E}} \tag{4.32}$$

Besides these errors, which occur because the operation is not defined for some operands, incompatible units can cause unit errors. Equation (4.23) and Equation (4.25) will fail if the units of left and right operand do not match.

4.2.2. The `PhysUnits` Extension

The proposed language extension is implemented in `de.ppme.physunits`. It follows closely the model outlined in Section 4.2.1 and is based on the physical units implementation `mps-example-physunits`[1] for the MPS BaseLanguage. The existing implementation was adopted to the PPME language for particle-mesh simulations and the domain-specific concepts.

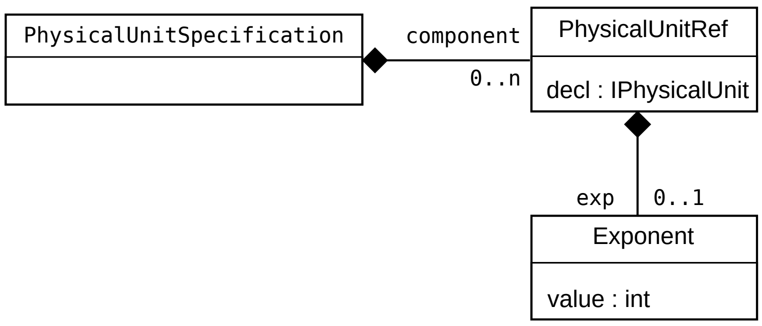


Figure 4.3.: Class diagram of the unit specification implementation in PPME.

A `PhysicalUnit` in PPME is a node with a name and a description implementing the `IPhysicalUnit` interface. It may also hold a unit specification to define a new derived unit (`PhysicalUnitSpecification`). Thus, `PhysicalUnit` relates to the unit declaration as presented above. These unit declarations are referenced by `PhysicalUnitRef`s, similar to constant references in programs. In Figure 4.3 the structure of a unit specification implementation is shown. We have modeled the specification node using a list of `component` references, abstracting the multiplication of (base or derived) units. Each unit reference has an option exponent `exp`. Please note that the choice of the exponent being an integer value was made for this proof of concept implementation. The drawbacks and possible solutions are discussed in Section 4.2.3.

As the physical units are essentially an extension of the type system the main logic of unit calculations and unit checking is located in type rules. To be specific, `overload_arithmeticOperations` is the operator overloading specification that holds the implementation of most of the presented rules. Listing 4.5 the implementation for Equation (4.25). First, the annotated specifications are expanded to base form (ll. 12–13). If the types match (l. 14) the inference for the operation type is performed in the original type system with the `operation type(..)` directive (l. 15). Finally, the resulting annotated type is constructed from the operation type and the simplified unit specification (ll. 16–20). In case the units do not match an error type is returned (l. 22).

[1]`https://github.com/fisakov/mps-example-physunits`

```
operation concepts: PlusExpression | MinusExpression
left operand type: <AbstractAnnotatedType()> is exact: false use strong subtyping true
right operand type: <AbstractAnnotatedType()> is exact: false use strong subtyping true
is applicable:
<no isApplicable>
operation type:
(operation, leftOperandType, rightOperandType)->node<>
  node<AnnotatedType> leftType = leftOperandType : AnnotatedType;
  node<AnnotatedType> rightType = rightOperandType : AnnotatedType;

  nlist<PhysicalUnitRef> leftSpec = PhysicalUnitConversion.expand(leftType.spec.component);
  nlist<PhysicalUnitRef> rightSpec = PhysicalUnitConversion.expand(rightType.spec.component);

  if (PhysicalUnitConversion.matching(leftSpec, rightSpec)) {
    node<> primtype = operation type(operation, leftType.primtype, rightType.primtype);
    nlist<PhysicalUnitRef> newSpec = PhysicalUnitConversion.simplify(leftSpec, operation.model.
         ↪ nodesIncludingImported(PhysicalUnitDeclarations));

    return <AnnotatedType(
      primtype: # primtype.copy : Type,
      spec: PhysicalUnitSpecification(component: # newSpec))>;
  } else {
    return <RuntimeErrorType(errorText: "units mismatch: " + leftOperandType + " and " + rightOperandType)>;
  }
}
```

Listing 4.5: Overloading rule for addition and subtraction of two operands with attached unit information.

The example uses several calls to the utility class `PhysicalUnitConversions` which contains reusable parts of unit calculations. The concept of unit expansion $\lceil u \rceil$ to the base form was already explained earlier. Specifications can contain derived units like mps which are obstructive when comparing units. Therefore, the `expand()` operation replaces all derived units, i. e., all units where the specification is not empty, by their specification until the specification contains only elementary units, i. e., is in base form.

```
 1  public static nlist<PhysicalUnitRef> expand(nlist<PhysicalUnitRef> spec) {
 2    list<[node<IPhysicalUnit>, int]> expanded = new arraylist<[node<IPhysicalUnit>, int]>(copy: spec.select({~unitRef
           ↪  => [unitRef.decl, unitRef.getExponent()]; }));
 3
 4    [node<IPhysicalUnit>, int] pair;
 5    do {
 6      pair = expanded.findFirst({~sp => sp[0] as PhysicalUnit.spec.component.isNotEmpty; });
 7      if (pair != null) {
 8        expanded.remove(pair);
 9        int exponent = pair[1];
10        expanded.addAll(pair[0] as PhysicalUnit.spec.component.select({~unitRef => [unitRef.decl, exponent * unitRef.
              ↪ getExponent()]; }));
11      }
12    } while (pair != null);
13
14    sequence<[node<IPhysicalUnit>, int]> expandedSorted = expanded.sortBy({~it => it[0].name; }, asc);
15    sequence<string> names = expandedSorted.select({~it => it[0].name; }).distinct;
16    list<[node<IPhysicalUnit>, int]> expandedFlatten = names.select({~name => expandedSorted.selectMany({~it =>
17        if (it[0].name :eq: name) { yield it; }
18      }).reduceLeft({~a,~b => [a[0], a[1] + b[1]]; }); }).toList;
19    expandedFlatten.removeWhere({~it => it[1] == 0; });
20
21    expandedFlatten.select({~it => it[1] != 1 ? <PhysicalUnitRef(
22        decl: # it[0],
23        exponent: Exponent(value: it[1]))> : <PhysicalUnitRef(
24        decl: # it[0],
25        exponent: null)>; }).toList;
26  }
```

Listing 4.6: Implementation of unit specification expansion (cf. Listing A.5, `expand()`)

Listing 4.6 shows the implementation of the expansion operation. First, the input specification is translated into a list of pairs, where each pair consists of a unit declaration and an integer exponent (l. 2). In lines 5–12 derived types are replaced by their specification iteratively until only base units are left. In the subsequent lines (ll. 14–19) the contained base units are combined so that each unit occurs at most once in the list. Recall that the unit specification stands for the multiplication of its contained units. Finally, the processed list of pairs is transformed back a specification, i. e., a list of unit references.

The `simplify()` operation in Listing 4.7 aims towards cancellation of units, e. g., in $mps \cdot s = m$. Like the expansion operation, `simplify()` works on a list of pairs, which consist of a unit declaration and an integer exponent. Technically, the unit specifications are first “demultiplexed” (ll. 10–30), that is the expansion of a unit specification in base form so that all unit references have $|exp| = 1$. For instance, $u = s^2$ will be expanded to $u' = s \cdot s$. After this expansion, all units that cancel each other out are removed from the list, and the remaining unit references are are multiplied, i. e., the exponents are added up (ll. 35–44).

```
 1  public static nlist<PhysicalUnitRef> simplify(nlist<PhysicalUnitRef> spec, nlist<PhysicalUnitDeclarations> decls) {

 8    do {
 9      done = true;
10      for (node<PhysicalUnit> cunit : sortedUnits) {
11        if (cunit.spec.component.isEmpty) { continue; }
12
13        sequence<[node<IPhysicalUnit>, int]> cspec = cunit.spec.component.selectMany({~ur => PhysicalUnitConversion.
             ↪ demultiplex([ur.decl, ur.getExponent()]); });
14        sequence<[node<IPhysicalUnit>, int]> cspecrecip = cunit.spec.component.selectMany({~ur =>
             ↪ PhysicalUnitConversion.demultiplex([ur.decl, -ur.getExponent()]); });
15
16        if (simplified.containsAll(cspec)) {
17          simplified.removeAll(cspec);
18          simplified.add([cunit, 1]);
19        } else if (simplified.containsAll(cspecrecip)) {
20          simplified.removeAll(cspecrecip);
21          simplified.add([cunit, -1]);
22        } else {
23          continue;
24        }
25
26        // start next iteration
27        done = false;
28        break;
29      }
30    } while (!done);

35    list<[node<IPhysicalUnit>, int]> simplifiedFlatten = names.select({~name => simplifiedSorted.selectMany({~it =>
36        if (it[0].name :eq: name) { yield it; }
37      }).reduceLeft({~a,~b => [a[0], a[1] + b[1]]; }); }).toList;
38    simplifiedFlatten.removeWhere({~it => it[1] == 0; });
39
40    simplifiedFlatten.select({~it => it[1] != 1 ? <PhysicalUnitRef(
41        decl: # it[0],
42        exponent: Exponent(value: it[1]))> : <PhysicalUnitRef(
43        decl: # it[0],
44        exponent: null)>; }).toList;
45  }
```

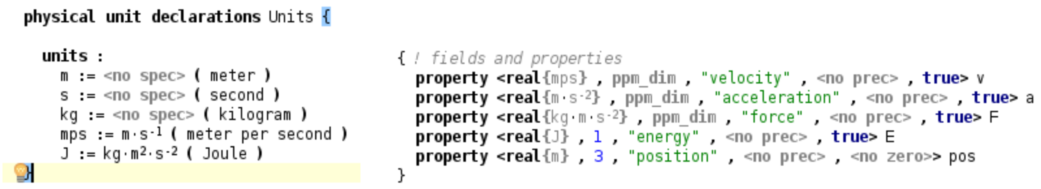
Listing 4.7: Implementation of the unit simplification `simplify()`. (cf. Listing A.5, `simplify()`)

An important part of the unit calculus in PPME is to decide whether two unit specifications are equivalent. The `matching()` function (see Listing 4.8) takes two specifications as input for which it assumes that they are in base form. The test simply transforms the input specifications into a list of pairs of declarations and integer exponents (ll. 2–5) and takes the disjunction of these two lists. If the disjunction is empty, i. e., there are no elements that occur only in one list, the two types match (l. 7).

```
1  public static boolean matching(nlist<PhysicalUnitRef> specA, nlist<PhysicalUnitRef> specB) {
2    list<[node<IPhysicalUnit>, int]> unwrappedA = new arraylist<[node<IPhysicalUnit>, int]>(copy: specA.select(
3      {~unitRef => [unitRef.decl, unitRef.getExponent()]; }));
4    list<[node<IPhysicalUnit>, int]> unwrappedB = new arraylist<[node<IPhysicalUnit>, int]>(copy: specB.select(
5      {~unitRef => [unitRef.decl, unitRef.getExponent()]; }));
6
7    unwrappedA.disjunction(unwrappedB).isEmpty;
8  }
```

Listing 4.8: Implementation of unit matching in `PhysicalUnitConversions` (cf. Listing A.5, `matching()`).

Integration in the Lennard-Jones Case Study We want to conclude this section with the integration of unit annotations into the Lennard-Jones case study. Figure 4.4 shows the actual unit declarations and annotations in the editor. With the annotations in place, the type checker is able to detect inconsistencies and report unit errors to the developer.



```
physical unit declarations Units {

  units :
    m := <no spec> ( meter )
    s := <no spec> ( second )
    kg := <no spec> ( kilogram )
    mps := m·s-1 ( meter per second )
    J := kg·m2·s-2 ( Joule )
}
```

```
{ ! fields and properties
  property <real{mps} , ppm_dim , "velocity" , <no prec> , true> v
  property <real{m·s-2} , ppm_dim , "acceleration" , <no prec> , true> a
  property <real{kg·m·s-2} , ppm_dim , "force" , <no prec> , true> F
  property <real{J} , 1 , "energy" , <no prec> , true> E
  property <real{m} , 3 , "position" , <no prec> , <no zero>> pos
}
```

Figure 4.4.: (Left) declaration of physical units for the Lennard-Jones case study. (Right) unit annotations on particle properties in the Lennard-Jones simulation program.

Given the specifications as above (plus `delta_t` being of type $[real, s]$), the following expression is verified to match the type and unit requirements.

$$\texttt{p}\rightarrow\texttt{pos = p}\rightarrow\texttt{pos + p}\rightarrow\texttt{v * delta\_t + 0.5 * p}\rightarrow\texttt{a * delta\_t}^2$$

Indeed, the assignment requires the right-hand side to be of type $Vector\langle[real, m]\rangle$. The sum consists of the three parts which all have to be of the result type $Vector\langle[real, m]\rangle$ according to Equation (4.25). In fact, $\texttt{p}\rightarrow\texttt{pos}$ trivially has the right type. For the second summand, we notice that the multiplication of velocity and time yields $m \cdot s^{-1} \cdot s = m$.

$$\frac{p \rightarrow v : Vector\langle[real, mps = m \cdot s^{-1}]\rangle \qquad \texttt{delta\_t} : [real, s]}{p \rightarrow v * \texttt{delta\_t} : Vector\langle[real, m]\rangle}$$

Finally, for the third summand the the multiplication of acceleration and time squared yields $m \cdot s^{-2} \cdot s^2 = m$ as well.

$$\frac{0.5 : real \qquad p \rightarrow a : Vector\langle[real, m \cdot s^{-2}]\rangle \qquad \texttt{delta\_t}^2 : [real, s^2]}{0.5 * p \rightarrow a * \texttt{delta\_t}^2 : Vector\langle[real, m]\rangle}$$

4.2.3. Evaluation of the Physical Unit Extension

The `PhysUnits` extension is a proof of concept incorporation of units of measure into PPME. Although the current status of the implementation is not feature complete, it already offers benefits in the development and debugging of simulation programs to improve quality and documentation. The annotation of physical units to data types as a modular language extension enables first-class support for units directly integrated in the type system. Moreover, it provides an additional abstraction without runtime cost for unit checking, paving the way for more sophisticated dimensional analysis. The unit information is only attached in the editor, where it is used for static analysis. Unit annotations are discarded during code generation, thus having no influence on the runtime of the simulation.

While supplying the developer with a powerful framework for simulation programs in physical sciences, the full freedom of specification is retained by the user. Unit declarations are not limited to some predefined set but can be freely chosen depending on the use case. Custom derived units, tailored for a specific domain, allow for concise notations and better readability.

We have shown that the implementation in the current state is powerful enough to fully annotate the Lennard-Jones case study with unit information. The successful unit check confirms the correctness of the implementation with regard to physical dimensions. Moreover, the unit annotations serve as in-line documentation. The purpose of particle properties such as velocity or acceleration is not only stated by a textual description but actually integrated into the type checking mechanism.

Open Tasks However, `de.ppme.physunits` has some pending issues. First, unit specifications are modeled using unit references with integer exponents. This choice forbids non-integer exponentiation with units as well as the square root operation. Instead, the system should use rationals as exponents in unit specifications, as suggested in Figure 4.5. Using rationals is a sensible compromise between accuracy and expressiveness.

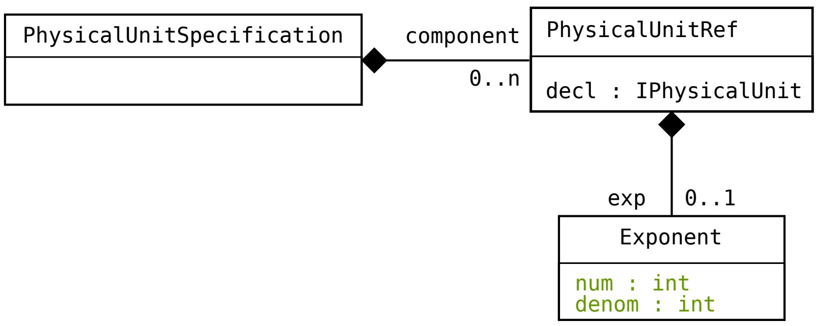


Figure 4.5.: Suggestion for an improved model of physical-unit specifications using unit references with rational exponent.

A second unimplemented feature are unit conversions and metric prefixes, that is, the automatic conversion of quantities by factors. The need for this arises from different measuring systems on the one hand, and the metric prefixes on the other hand. Differing systems of measurement use diverse units for the same class of measure, e. g., foot and meter both depicting a length. The unit engine should be able to detect compatibility of units within some class and infer the correct conversion by a factor. Similarly, with a list of available prefixes and corresponding factors conversions between cm and m can be left to the system.

Finally, the unit engine might be generalized so that overloaded operation rules can be simplified. Given an implicit conversion $e : \tau \mapsto e : [\tau, \mathbf{1}]$ which adds a unitless annotation to some expression, the unit engine can assume to always deal with annotated types. Thus, case distinctions for operations where only one or both operands have unit annotations can be dismissed in favor of a uniform unit resolution rule.

5. Numerical Optimizations

Applications in the fields of science and engineering often depend on floating-point arithmetic in calculations to approximate real arithmetic. As numerical operations are such an essential part of scientific computations they are a common target for optimizations. In many cases, program transformations on loops are used to find a good space-time-tradeoff and improve data locality. Examples for such transformations are loop fusion, splitting, and unrolling [LW91; Lup+15].

In this work, we focus on numerical optimizations based on transformations of floating-point expressions. These can be used to improve a program's *performance*, its *accuracy*, or both. Such optimizations often rely on an *abstract semantic* for the arithmetic expressions. The abstract semantic governs program transformations and ensures semantic equivalence of the resulting program. It allows to build up equivalent graphs and inspect them with regard to the desired property, e. g., accuracy or performance. However, a naïve implementation of a search over all equivalent expressions would result in a combinatorial blowup, as pointed out by Ioualalen and Martel [IM12].

Section 5.1 introduces the notion of (abstract) program equivalence graphs as a foundation for numerical transformations. It shortly explains the characteristics of floating-point operations and describes the principles of equivalence graphs for efficient search algorithms. In section 5.2, a tool for for automatic improvement of floating-point accuracy is integrated into PPME in order to help the developer find problematic arithmetic expressions and improve their computation. It uses a database of applicable rules and a heuristic to build a search space and find an improved computation scheme for a given floating-point expression. Contribution [C3], the investigation and evaluation of numerical optimizations, is addressed by integrating the external tool and elaborating its impact on the implemented case studies.

5.1. Program Equivalence Graphs

The notion of an *Abstract Program Equivalence Graph* (APEG) denotes an intermediate representation to describe a large set of equivalent expression efficiently [IM12]. APEGs are inspired by the Equivalence Program Expression Graphs (EPEGs) intermediate representation which is used to represent equivalent versions of imperative programs [Tat+11]. EPEGs were originally defined to model the effects of different optimization stages, i. e., program transformations, in a non-destructive way. Instead of using the result of one transformation as input for the next phase, the possible transformations are maintained as equality information on the intermediate representation (IR).

Based on the idea of exploring equivalent rewrites, Ioualalen and Martel aim to provide a tool for automatic improvement of accuracy of expressions at compile time. Therefore, they propose an approach in two phases. The first phase is the construction of the search space of equivalent expressions. In the second phase, the constructed space is explored with regard to some metric to find a replacement for the original expression with better (ideally, the best) accuracy.

The remainder of this sections is structured as follows. Firstly, we present the formal definition of an APEG in section 5.1.1, introducing equivalent nodes classes and abstraction boxes in the IR. Secondly, the construction of APEGs from an initial expression is elucidated in section 5.1.2. Finally, section 5.1.3 covers the exploration of the originating graph.

5.1.1. Formal Definition

In general, the problem with equivalences for arithmetic expressions is the exponential blowup, even for small expressions. For instance, a sum of ten terms yields almost 40 million evaluation schemes. Thus, APEGs are designed to represent an exponential set of equivalent expressions in polynomial space. However, it should be noted that not all equivalent expressions are covered due to a polynomial under-approximation in the APEG construction.

An APEG is always based on some initial expression e and forms an abstract representation of an exponential number of equivalent expression in polynomial size. It contains usual arithmetic operators (like $+$, $\times$, or $-$), constants, and variables with a range interval provided by the user (e. g., $[3.14; 3.15]$). Throughout the paper only unary and binary operators are considered. The syntax tree of the expression is extended by means to describe equivalences of nodes in the tree. Therefore, two particular extensions were made to classical syntax trees.

Classes of Equivalent Nodes Each node in the APEG can be attached with alternatives that are equivalent with respect to certain equality relations $\triangleright = \{\triangleright_i, 1 \leq i \leq n\}$. $\triangleright$-equality of two expressions e_1 and e_2 is defined via the transitive reflexive closure of these binary relations ([IM12, Def. 1]). To achieve mathematical equivalence, $\triangleright$ is often defined as (a subset of) the rules on real arithmetic, i. e., associativity, commutativity, etc. This implies that nodes in a fully expanded APEG are classes of expressions equivalence with regard to $\triangleright$. To keep the space requirement small common parts in expression are reused across an APEG.

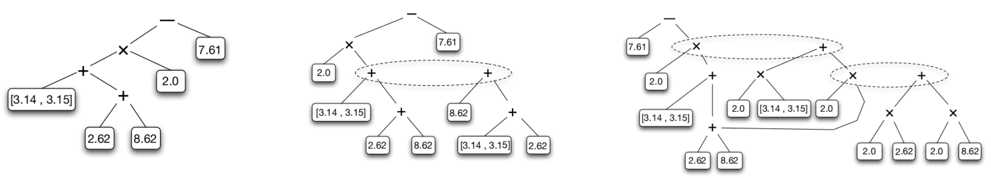


Figure 5.1.: (Left) Syntactic tree of the expression e. (Middle) APEG built with associativity. (Right) APEG built with product propagation and reuse of sub-expressions [IM12, Figures 1–3].

Figure 5.1 illustrates the enrichment of the syntax tree of an expression

$$e = ([3.14; 3.15] + (2.62 + 8.62)) \times 2.0 - 7.61$$

with rules of associativity and product propagation. In the second graph an equivalent expression for the addition is found through associativity and is attached to original node, forming an equivalence class (indicated by the dashed ellipse). In the third graph, product propagation was used to find an equivalence for the multiplication node. Note that the common part $(2.62 + 8.62)$ occurs only once in the APEG.

Abstraction Boxes Abstraction boxes are abstractions of sub-trees where the same operator is uniformly applied to a set of sub-expressions. The abstraction box is a placeholder for all possible parsings that can be obtained for the sub-expressions and the specific operator. Given the box $\boxed{+, (a, b, c)}$, the abstraction stands for every possible way to sum the values for a, b, and c. This approach allows for space efficiency whilst retaining expressiveness of an exponential number of equivalent expressions in the APEG.

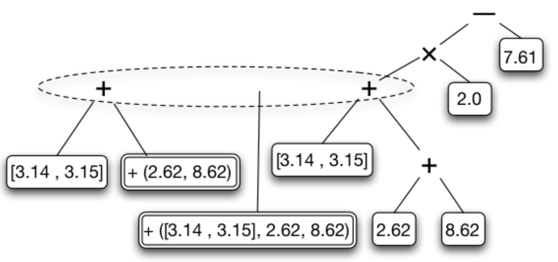


Figure 5.2.: Enrichment of an APEG with abstraction boxes [IM12, Figure 4].

The APEG in fig. 5.2 contains two abstraction boxes (rectangles with double outline). In this case, the boxes represent all summation schemes for the box elements which are yielded by the equivalent transformations, i. e., associativity and commutativity of addition.

5.1.2. Construction

As stated above, APEGs are built upon an initial expression e. Construction transformations are used to expand the initial syntax tree by attaching new nodes to equivalence classes with respect to $\triangleright$ and grouping sub-expressions into abstraction boxes. These transformations only add nodes or boxes but never discard elements from the graph. Both transformations, homogenization and expansion, are designed to be executed after one another.

Homogenization In order to cover as many expressions with abstraction boxes as possible (i. e., produce the largest abstraction boxes) the notion of *homogeneous* expressions is useful. An APEG is *fully homogeneous* if it consists only of variables or constants and some symmetric binary operator $*$, e. g., $e = a + (b + c)$ is fully homogeneous. *Partial homogeneity* is a weaker formulation as it only requires the APEG to contain a fully homogeneous sub-expression where the operands are variables, constants, or sub-expressions. For instance, $e' = ((a \times b) + (c + d)) \times e$ is partially homogeneous where $e_1 + (c + d)$ is a fully homogeneous sub-expression of e' for $e_1 = a \times b$.

The goal of homogenization transformations is to add new nodes to the APEG to introduce homogeneous sub-expressions. The exact set of applicable transformations depends on the operator to evaluate. Ioualalen and Martel exemplarily describe the homogenization for multiplication and the unary minus operator.

Expansion Expansions functions aim to insert abstraction boxes with as many operands as possible. Therefore, expansion is performed on each node of an APEG recursively. Ioualalen and Martel describe three types of expansion functions, *horizontal expansion*, *vertical expansion*, and *box expansion*. Horizontal expansion splits a homogeneous sub-tree and and introduces a new abstraction box for the leaves of the left or right part. Thus, for each binary operator there are two abstraction boxes that can be built.

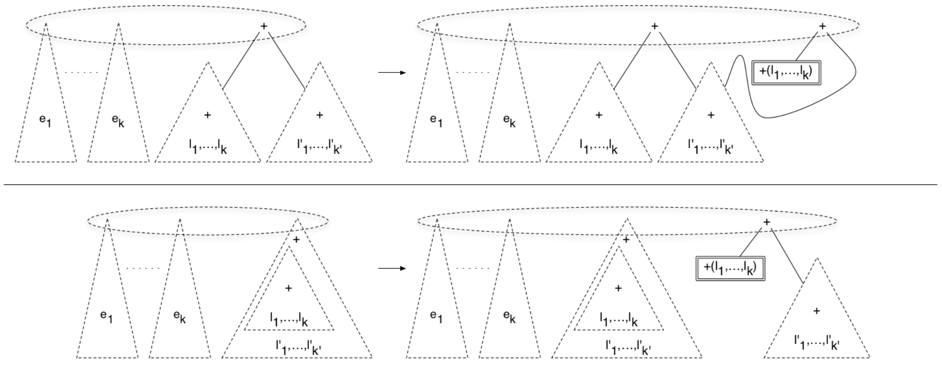


Figure 5.3.: (Top) Horizontal expansion of the left part of an addition. (Bottom) Vertical expansion of the sub-expression containing $l_1 \dots l_k$ [IM12, Figure 7].

Vertical expansions splits a homogeneous part at a sub-expression. Similarly to the horizontal expansion, an abstraction box can be built for the leaves of the split-off sub-expression or the leaves of the enclosing expression. Figure 5.3 illustrates one possible expansion for both horizontal and vertical expansion.

Finally, box expansion is used to merge nested abstraction boxes with the same operator. The box expansion step grows the abstraction boxes produced through horizontal and vertical expansion.

5.1.3. Exploration

A fully expanded APEG represents an exponential amount of equivalent expressions. Once this search space is constructed for a given expression e it can be explored to find a better expression with respect to some optimization objective. The approach described by Ioualalen and Martel aims to minimize the roundoff error of an expression and thus improve the accuracy.

The minimization problem is defined over a custom arithmetic where error terms are attached to the floating-point values [IM12; Mar09]. These error terms describe the range in which the rounding error lies. Ioualalen and Martel advise to use an external library for error computation with arbitrarily large precision. In particular, a value x in the arithmetic is represented as $x = (f_x, e_x)$ where f_x is the floating-point number approximating x and e_x is the rounding error of this approximation. Furthermore, $\circ(v)$ denotes the rounding of the value v, and $\varepsilon(v)$ denotes the roundoff error occurring when rounding v to $\circ(v)$. Thus, with $\varepsilon(v) = v - \circ(v)$, the error evaluation of the basic operations is as follows:

$$x + y = (\circ(f_x + f_y), e_x + e_y + \varepsilon(f_x + f_y)) \tag{5.1}$$

For multiplication, $(f_x + e_x) \times (f_y + e_y)$ is expanded for the error calculation:

$$x \times y = (\circ(f_x \times f_y), f_x \times e_y + f_y \times e_x + e_x \times e_y + \varepsilon(f_x \times f_y)) \tag{5.2}$$

Note that error terms are propagated among computations. Other operators, such as division and square root, are developed using power series to compute the propagation of the error [Mar06]. In general, the quality of the determined error terms influences the search space exploration.

For single expressions the computation of errors is simple, and finding the one with minimal error among a set of expressions is trivial. However, the concept of equivalence classes can cause combinatorial explosion when evaluating operations $*(p_1, p_2)$, where p_1 and p_2 are equivalence classes. Therefore, Ioualalen and Martel restrict the exploration to a limited depth search. The evaluation of an expression is conducted considering only the best evaluation of it's sub-expressions (local choice strategy). For abstraction boxes a greedy algorithm is used to synthesize an accurate, yet not necessarily optimal, expression. The algorithm greedily merges expressions $*(p_i, p_j)$ with minimal error into a new term p_{ij}, removing the expression. The final node left corresponds to the root of the synthesized expression for the box.

Overall, this techniques allows to explore the search space spanned by an APEG in polynomial time. Experimental results of Ioualalen and Martel haven shown improvements in accuracy by an average of 50% [IM12].

5.2. Herbie — Automatically Improving Floating-Point Accuracy

The major drawback of techniques for mitigating rounding errors in floating-point expressions, as described in the numerical methods literature, is that they have to be applied manually. Typically, these techniques require a deeper understanding of the details of floating-point arithmetic as well as mathematical expertise.

Herbie is an attempt to enable automatic floating-point arithmetic improvements by discovering the expression rewrites experts find and perform to increase accuracy [Pan+15]. It uses a heuristic search to estimate and localize rounding errors by using sample points. To improve a given expression, it applies a database of rules, takes series expansions, and combines improvements for different input regions.

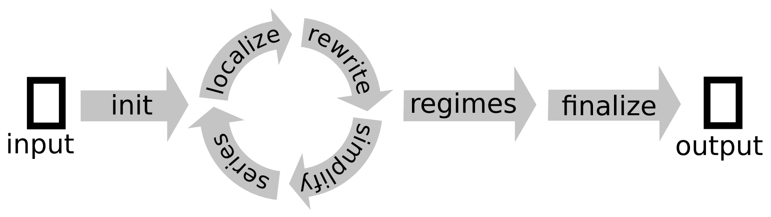


Figure 5.4.: Herbie's process for improving program accuracy [Pan+15, Figure 1].

Figure 5.4 illustrates Herbie's process for improving a program's accuracy [Pan+15]. After initialization, the tool localizes floating-point inaccuracies through sample point computations. Once inaccuracies are found, Herbie attempts to avoid them by modifying the program using rewrites. The error elimination is carried out by applying rewrite rules describing basic arithmetic artifacts. Thereafter, Herbie performs a simplification pass to cancel terms with the general goal to produce a smaller, equivalent program. In case rounding errors are detected for inputs around zero or near infinity for which no better program could be found using rewrite rules, Herbie performs series expansions to approximate the result. This often helps to avoid over- and under-flows.

This core loop of actions is iteratively executed, where each iteration yields candidate programs. Between iterations the set of candidates is pruned to keep only the programs which achieve best accuracy at least at one sample point. As in most cases not a single candidate program is most accurate for all input points, Herbie assembles a final program in a regime-inference pass. Regimes correspond to input regions for which different expressions are most accurate. In order to prevent over-fitting, regime inference has to find a trade-off for the number of branches to apply. Hence, Herbie combines several candidate programs to achieve an improvement of accuracy over all input points.

5.2.1. Integration into PPME

Herbie support is implemented as a *plug-in solution* within MPS. This enables for hooking up the external analysis in PPME projects for user selected expressions. Since Herbie's analyses are computational expensive, expressions are not inspected automatically but must be flagged for evaluation by the user. PPME offers an intuitive way to mark expressions utilizing the context menu when right-clicking on a code fragment. Once all desired expressions are marked the user can trigger the analysis and transformation process for the active editor. The result of the external execution is then annotated to the code fragments.

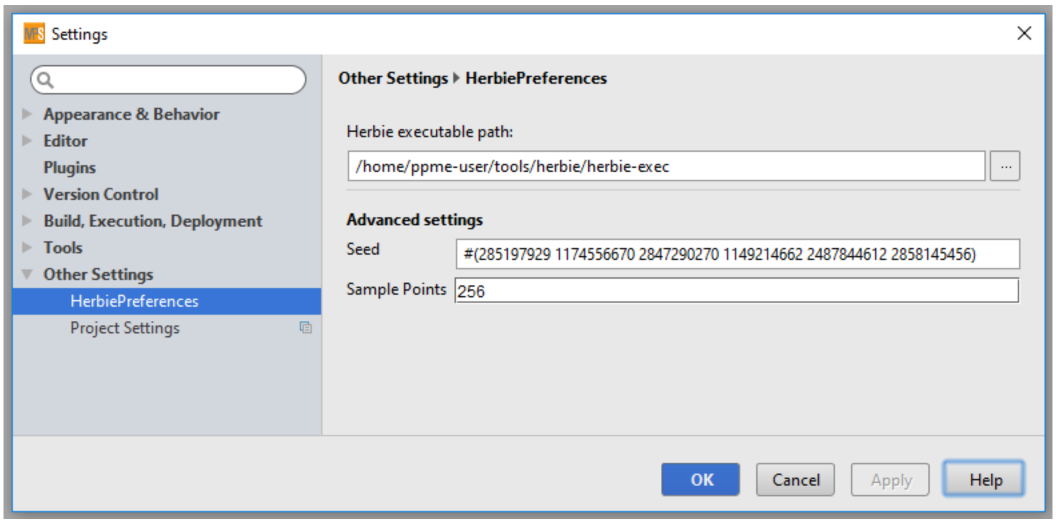
Requirements We use a develop build version of Herbie[1] for the integration. In order for the plug-in to be able to execute Herbie, the Herbie's requirements have to be fulfilled. In particular, this means that Racket 6.4 or higher has to be installed and that Herbie's `herbie/interfaces/inout.rkt` interface has been compiled to a single executable. Besides that, the plug-in implementation contains all other dependencies for execution.

Settings The Herbie plug-in offers a settings dialog which allows to configure the plug-in's behavior as shown in Figure 5.5. It can be found via `File > Settings ↪ > Other Settings > Herbie` in the MPS settings. Most importantly, the user has to select the Herbie executable used for the analyses. The path to the file can comfortably be chosen using a native dialog for the user's operating system. Note that the plug-in integrates seamless with the develop environment.

The advanced settings section provides additional control over Herbie's behavior. For instance, the seed value influences the selection of random sample points Herbie evaluates candidate programs on. The format for the seed is that of Racket's `vector->pseudo-random-generator!`. Setting the seed explicitly can be used to make Herbie's results reproducible between different runs. The number of sample points determines the number of randomly-selected points annotated expressions are evaluated on. The default value of 256 give sufficient results for most programs, higher values for more sample points will slow down Herbie.

The Herbie Executable In its current state, Herbie offers two ways to check expressions for improvements. The first is to run a full-fledged analysis on a series of problem descriptions. This will process all specified problems at once and will produce an HTML output report. Besides general information on input/output error and improved regime the results are visualized with an arrow notation (showing the improvement in accuracy in bits) and graphs for the error distribution. The information is stored in an accessible form as JSON data, the visualizations are produced in PNG format.

[1] `https://github.com/uwplse/herbie`, commit hash `f6ebaea`



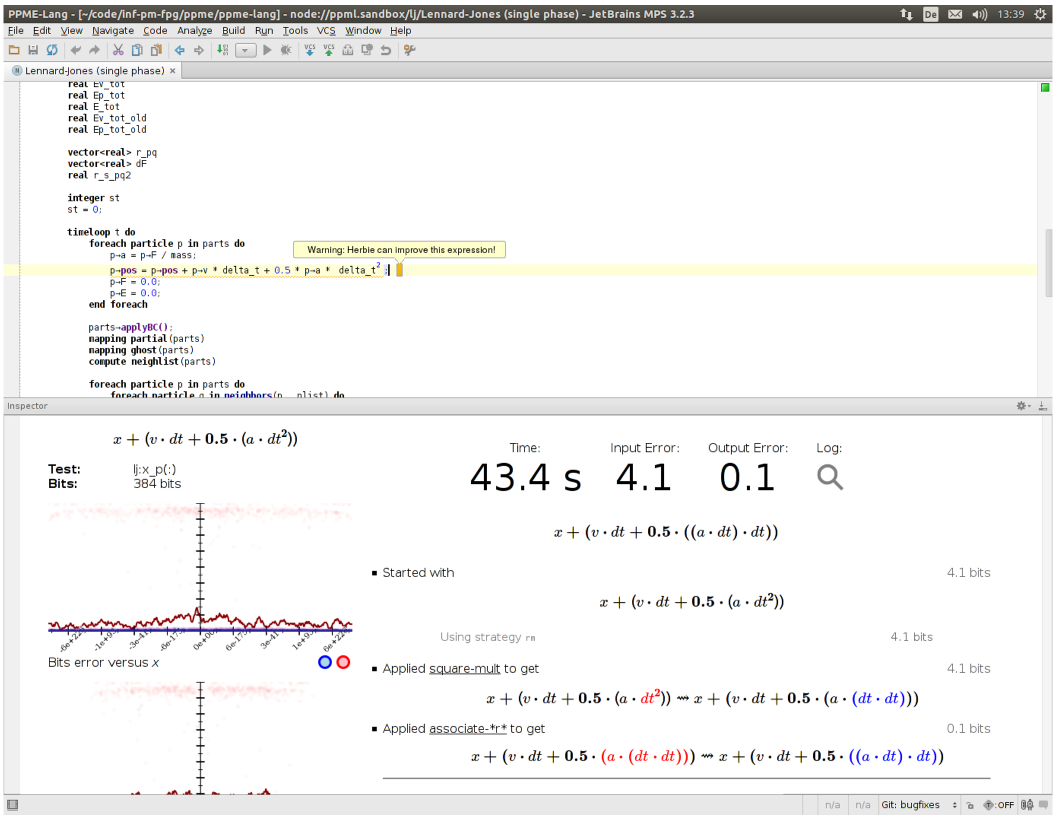
Herbie executable path: /home/ppme-user/tools/herbie/herbie-exec

Advanced settings

Seed: #(285197929 1174556670 2847290270 1149214662 2487844612 2858145456)

Sample Points: 256

Figure 5.5.: Configuration panel for the Herbie integration in PPME.


PPME-Lang - [~/code/inf-pm-fpg/ppme/ppme-lang] - node://ppml.sandbox/lj/Lennard-Jones (single phase) - JetBrains MPS 3.2.3
File Edit View Navigate Code Analyze Build Run Tools VCS Window Help
Lennard-Jones (single phase)
Inspector
Bits error versus x
n/a
n/a
Git: bugfixes

```
        real Ev_tot
        real Ep_tot
        real E_tot
        real Ev_tot_old
        real Ep_tot_old

        vector<real> r_pq
        vector<real> dF
        real r_s_pq2

        integer st
        st = 0;

        timeloop t do
            foreach particle p in parts do
                p→a = p→F / mass;
                p→pos = p→pos + p→v * delta_t + 0.5 * p→a *  delta_t^2 ;
                p→F = 0.0;
                p→E = 0.0;
            end foreach

            parts→applyBC();
            mapping partial(parts)
            mapping ghost(parts)
            compute neighlist(parts)

            foreach particle p in parts do
                foreach particle q in neighbors(p, nlist) do
```

Warning: Herbie can improve this expression!

$x + (v \cdot dt + \mathbf{0.5} \cdot (a \cdot dt^2))$

Test: lj:x_p(:)
Bits: 384 bits

Time:	Input Error:	Output Error:	Log:
43.4 s	4.1	0.1	🔍

$$x + (v \cdot dt + \mathbf{0.5} \cdot ((a \cdot dt) \cdot dt))$$

- Started with — 4.1 bits

$$x + (v \cdot dt + \mathbf{0.5} \cdot (a \cdot dt^2))$$

Using strategy rm — 4.1 bits

- Applied square-mult to get — 4.1 bits

$$x + (v \cdot dt + \mathbf{0.5} \cdot (a \cdot dt^2)) \rightsquigarrow x + (v \cdot dt + \mathbf{0.5} \cdot (a \cdot (dt \cdot dt)))$$

- Applied associate-*r* to get — 0.1 bits

$$x + (v \cdot dt + \mathbf{0.5} \cdot (a \cdot (dt \cdot dt))) \rightsquigarrow x + (v \cdot dt + \mathbf{0.5} \cdot ((a \cdot dt) \cdot dt))$$

Figure 5.6.: Mockup of Herbie's report integrated into PPME.

The second variant is to use Herbie's command line interface which is called with a single problem instance. Execution output is written directly to the standard output with information limited to input/output error and the improved regime. Thus, the interface is lightweight as it does not produce as detailed reports as the full-fledged analysis. Furthermore, the interface has to be set up only once and can then be reused for multiple problem instances, whereas the report generation requires to initialize Herbie anew before each expression to check. Additionally, usage of the command line interface prevents problems with file read and write permissions.

Listing 5.1 shows the output structure for some test case. The first line shows the (pseudo-) random seed used to generate the sample points. Lines two and three show the input and output error of the floating-point arithmetic compared to the exact result, respectively. The improved regime for a given input expression will be contained in the fourth line.

```
; Seed: #(285197929 1174556670 2847290270 1149214662 2487844612 2858145456)
; Input error: 4.15986215899212
; Output error: 0.0703125
(λ (...) (...))
```

Listing 5.1: Example of Herbie's command line output.

We decided to use the lightweight interface for the integration into PPME due to simpler invocation and parsing of the result. In addition, the command line interface is successfully used for the integration of Herbie into compilers. Both the GHC plug-in[2] and the Rust Linter plug-in[3] rely on the lightweight interface. This decision allows for a simple implementation of the processing pipeline. On the other and, it shortcuts issues with input file generation, parsing the output report and integrating the report fragments. However, it should be noted that conceptually the full report with its visualization can be integrated using a similar approach.

Both variants require the input to be in the form of a `herbie-test` in Racket. The syntax for a Herbie test case is given as follows:

```
(herbie-test (<variables>) "<name>" <expression>)
```

Each test has an arbitrary name field (`<name>`). Commonly, a combination of filename and line number of the expression to test is used. Since simulations written in PPME are no traditional text files using the node id of the root expression to test is a sane choice. As a benefit, this allows to map a given test case to a specific node in the PPME program.

Besides the name tag, a Herbie test case is very similar to a pure lambda expression in Racket

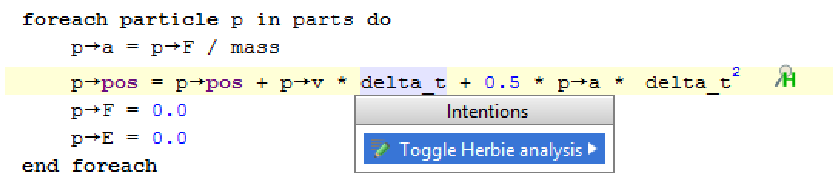
```
(lambda (<variables>) <expression>)
```

where all free variables in the expression `<expression>` are listed in the variables block `<variables>`. The free variables of the lambda expressions are the variables to test in the Herbie execution, i. e., these variables are tested with a series of sample points.

Generating Herbie Test Cases The user can annotate expressions in the editor window of PPME to mark them for the next analysis via the intentions dialog (Alt + Enter). The intention will always annotate the root of the expression the cursor is currently placed on. A small icon indicates that an expression is marked for analysis.

[2]https://github.com/mikeizbicki/HerbiePlugin

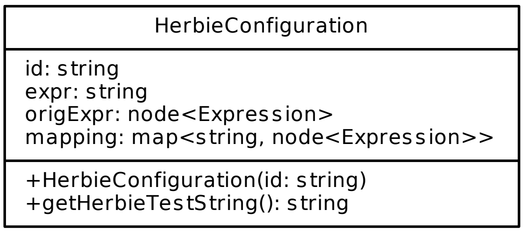
[3]https://github.com/mcarton/rust-herbie-lint



```
foreach particle p in parts do
    p→a = p→F / mass
    p→pos = p→pos + p→v * delta_t + 0.5 * p→a * delta_t²
    p→F = 0.0
    p→E = 0.0
end foreach
```

Figure 5.7.: The editor intention to toggle the Herbie analysis on the selected expression. The icon at the end of the line indicates that the expressions is currently marked for analysis.

Expression nodes are translated into a configuration object (`HerbieConfiguration`) by a translator utility class (`ExpressionTranslator`). In short, all expressions are transformed into a prefix notation matching the Racket syntax, where operations such as exponentiation and (square-) roots are translated into the corresponding functions supported by Herbie.



HerbieConfiguration
id: string
expr: string
origExpr: node<Expression>
mapping: map<string, node<Expression>>
+HerbieConfiguration(id: string)
+getHerbieTestString(): string

Figure 5.8.: The `HerbieConfiugartion` class which contains the information necessary to start the analysis.

At the same time, a variable table of string identifiers pointing to expression nodes is created for the translated expressions. PPME treats every variable reference and particle access as an input variable for the test case. All variable and property references add a mapping from their variable name to themselves. For arrow expressions (access on particles or particle lists) a (unique) identifier is created, and a mapping from this identifier to the arrow expression is added to the table.

The configuration class (as shown in Figure 5.8) contains the translated expression as string `expr`, a reference to the original expression node and its identifier (`origExpr` and `id`), as well as the variable table `mapping`. For the expression highlighted in fig. 5.7 the following test case is generated:

```
(herbie-test (p_pos p_a delta_t p_v)
  "2430378650379961582"
  (+ (+ p_pos (* p_v delta_t)) (* (* 0.5 p_a) (expt delta_t 2))))
```

The variable reference `delta_t` is directly translated to a variable for optimization. For the particle accesses ($p \to pos$, $p \to a$, and $p \to v$) corresponding identifiers are created and added to the list of optimization variables. Table 5.1 shows the mappings of PPME expressions to Herbie notation, where a and b are arbitrary expressions, x is a variable reference, p is a particle, ps is a particle list, and y is a vector.

The name of the test case is derived from the root expression's identifier in the program structure. As described above, the expression itself is translated into prefix notation. Note that the exponentiation `delta_t`^{2} in PPME is translated into a Racket function `(expt delta_t 2)`.

PPME notation	Herbie notation
$-a$	`(- a)`
$\sqrt{a}$	`(sqrt a)`
x	`x`
$p \rightarrow pos$	`p_pos`
$ps \rightarrow v$	`ps_v`
$y[0]$	`y_0`

PPME notation	Herbie notation
$a + b$, $a - b$	`(+ a b)`, `(- a b)`
$a * b$, a/b	`(* a b)`, `(/ a b)`
a^b	`(expt a b)`
$a == b$	`(= a b)`
$a < b$, $a > b$	`(< a b)`, `(> a b)`
$a <= b$, $a >= b$	`(<= a b)`, `(>= a b)`

Table 5.1.: (Left) Translation of unary operators and variable references. (Right) Translation of binary operators.

Running Analyses Triggering a Herbie analysis will process all annotated expressions in the active editor. The analysis action can be found in the right-click context menu of the editor and is labeled “Run Herbie Analysis”. By clicking on the context menu entry the user starts a series of actions and transformations. Figure 5.9 illustrates how the involved parts operate with one another with an interaction diagram. The whole process is steered by `HerbieAction`. Overall, it is responsible for fetching the nodes to test, run the analysis on the node, and write the result back to the node.

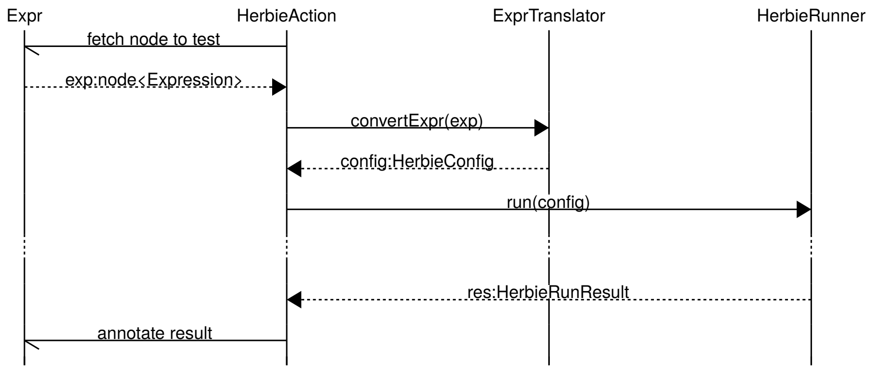


Figure 5.9.: Sequence diagram for the execution of a Herbie analysis for a single expression.

First, the node to test is fetched from the PPME model. From this node a configuration object (`HerbieConfiguration`) is created by the `ExpressionTranslator` utility class as described in section 5.2.1. The configuration object is used to feed the test case to Herbie. Responsible for this step is the `HerbieRunner` which takes the configuration, runs the Herbie executable with the test case as input, and waits for the process to terminate. The result of the execution is summarized in a `HerbieRunResult` object (see fig. 5.10). The generic design of the execution class (`Runner`) and the result container (`RunResult`) allows to reuse the same setup for other tools by extending the existing framework.

Given the result object the original expression node is annotated with additional information, i. e., the input and output error, and the optimized computation regime for the arithmetic expression. This approach allows to inspect the result of a check performed by Herbie, and it keeps the original expression in place without modifying it. In fig. 5.11 the inspection window for an annotated node is shown (Alt + 2). All information given by Herbie is displayed and the user is able to see the effect of the transformation.

Instead of transforming the original node into target source code the annotated optimization has to be generated. The generators of PPME are responsible for replacing

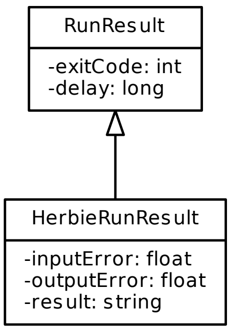
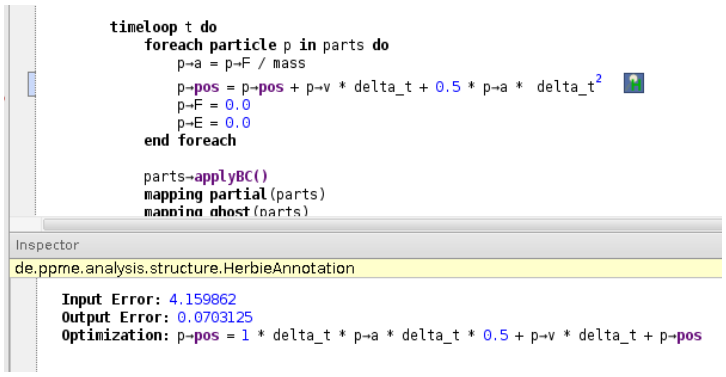


Figure 5.10.: The `HerbieRunResult` class and its super class `RunResult`. The result of an external Herbie execution is stored in the data class.



```
timeloop t do
    foreach particle p in parts do
        p→a = p→F / mass
        p→pos = p→pos + p→v * delta_t + 0.5 * p→a *  delta_t²
        p→F = 0.0
        p→E = 0.0
    end foreach

    parts→applyBC()
    mapping partial(parts)
    mapping ghost(parts)
```

Inspector

de.ppme.analysis.structure.HerbieAnnotation

```
Input Error: 4.159862
Output Error: 0.0703125
Optimization: p→pos = 1 * delta_t * p→a * delta_t * 0.5 + p→v * delta_t + p→pos
```

Figure 5.11.: The inspection view on an annotated expression with the results of a Herbie analysis.

the annotated node by the optimized regime before the text generation phase. Note that analysis and code generation are separate processes, where the Herbie analysis is one example for integrating external tools in the simulation development.

Parsing Herbie's Output Parsing Herbie's output (see listing 5.1) is necessary to integrate the result back into PPME. For this purpose, a small utility library called `herbie-parser`[4] is used. The library consists of an ANTLR 4 grammar from which lexer and parser for Herbie's output are generated [Par13], as well as a simple interface for calling the parser. Moreover, an (empty) visitor class is generated and can be extended to process the parsed output. Listing 5.2 shows the ANTLR grammar which describes the syntax of Herbie's output. The grammar itself is very general and leaves interpretation of the parsed expression to the application using it.

In PPME, the transformation from Herbie back to the internal format of expressions is handled by `HerbieOutputVisitor`, an implementation of the visitor class generated by ANTLR. The visitor basically performs the inverse transformation of `ExpressionTranslator`, taking a Racket application and turning it into PPME nodes. Table 5.2 lists a selection of transformation rules. Some of the arithmetic expression natively supported in Herbie/Racket do not have a strictly corresponding equivalent in PPME. For instance, `square` and `cube` are translated to generic exponentiation by factor. Variable occurrences in the Racket output are transformed by using the lookup table which was generated alongside the test case (refer to section 5.2.1).

[4] `https://bitbucket.org/ppme/herbie-parser`

```
grammar Herbie;

expression    : constant | if_statement | variable | application ;
application   : "(" op=expression expression* ")" ;
if_statement  : "(" "if" condition=expression thenBranch=expression elseBranch=
    ↪ expression ")" ;
variable      : IDENTIFIER ;
constant      : BOOLEAN | NUMBER ;

IDENTIFIER    : (INITIAL SUBSEQUENT*) | "+" | "-" ;
BOOLEAN       : "t" | "f" ;
NUMBER        : SIGN? (UINT | UFLOAT) (E SIGN? UINT)? ;

fragment INITIAL    : LETTER | "_" | "*" | "/" | "<" | "=" | ">" ;
fragment SUBSEQUENT : INITIAL | DIGIT | "." | "+" | "-" ;
fragment LETTER     : [a-zA-Z] ;
fragment UINT       : "0" | [1-9]DIGIT* ;
fragment UFLOAT     : DIGIT+ "." DIGIT* | "." DIGIT+ ;
fragment E          : "e" | "E" ;
fragment SIGN       : "-" | "+" ;
fragment DIGIT      : [0-9] ;

WS : [ \t\n\r]+ -> skip ;
```

Listing 5.2: ANTLR grammar specification for Herbie's output format.

Herbie notation	PPME notation
`(+ a b), (- a b)`	$a + b$, $a - b$
`(* a b), (/ a b)`	$a * b$, a/b
`(square a), (cube a), (expt a b)`	a^2, a^3,a^b
`(sqrt a)`	$\sqrt{a}$
`(= a b), (< a b), (> a b)`	$a == b$, $a < b$, $a > b$
`(<= a b), (>= a b)`	$a <= b$, $a >= b$
`x`	`lookup(x)`

Table 5.2.: Translation of Herbie output to PPME expressions, where a and b are arbitrary expressions, and x is a variable.

Recall that Herbie outputs either a single expression optimized for accuracy, or a more complex regime with conditional branching. The former is a simple replacement for the right-hand side of the original assignment.

The latter requires to replace the original expression statement with the conditional regime. Therefore, the conditional branching is translated into if-then-else-statements in PPME, where each branch contains a single assignment statement for the original expression. In case the expression to optimize is the initialization of a variable declaration, the transformation has to ensure that declaration and initialization are split.

In listing 5.3 an exemplary analysis call and its corresponding translation to PPME is shown. The computed optimizing regime replaces the following assignment statement of the Lennard-Jones case study:

$$dF = (24.0 \cdot \varepsilon \cdot r_{pq}) \cdot (2.0 \cdot \frac{\sigma^{12}}{r_{s_{pq}}^{7}} - \frac{\sigma^{6}}{r_{s_{pq}}^{4}})$$

Herbie is able to improve the accuracy significantly, reducing the error of 34 bits to slightly more than 15 bits. The regime intervals depend on the value of $r_{s_{pq}}$, i. e., the squared distance of particles p and q. For each of the three cases an appropriate optimization is selected. Most notably, Herbie expands the exponention of $sigma$ to a series of multiplications in every case.

```
(herbie-test (epsilon sigma dF r_pq r_s_pq2)
  "7068769801678688643"
  (* (* (* 24.0 epsilon) r_pq) (- (* 2.0 (/ (expt sigma 12) (expt r_s_pq2 7))) (/ (expt sigma 6) (expt r_s_pq2 4)))
       ↪ ))

; Input error: 34.03036766696533
; Output error: 15.641861746224407

(λ (epsilon sigma dF r_pq r_s_pq2)
  (if (< r_s_pq2 -3.4035520833325653e-94)
    (* (- (* epsilon 24.0)) (cube (/ (cbrt (* (* sigma (* sigma (* sigma (* sigma (* sigma sigma))))) r_pq)) (cbrt
         ↪ (* r_s_pq2 (* r_s_pq2 (* r_s_pq2 r_s_pq2)))))))
    (if (< r_s_pq2 1.461628127388658e-31)
      (/ (* (* (* sigma (* sigma (* sigma (* sigma (* sigma sigma))))) r_pq) (* (- epsilon) 24.0)) (* (/ 1 r_s_pq2)
           ↪  (* (/ 1 r_s_pq2) (* (/ 1 r_s_pq2) (/ 1 r_s_pq2)))))
      (* (- (* epsilon 24.0)) (/ (* sigma (* sigma (* sigma (* sigma (* sigma sigma))))) (/ (* r_s_pq2 (* r_s_pq2
           ↪ (* r_s_pq2 r_s_pq2))) r_pq))))))
```

```
Input Error: 34.03037
Output Error: 15.641862
Optimization:
  if (r_s_pq2 < -3.4035520833325653E-94) then
    dF = ((-(epsilon * 24.0)) * ( (( ((sigma * (sigma * (sigma * (sigma * (sigma * sigma))))) * r_pq)^{1/3}) / ((
         ↪ r_s_pq2 * (r_s_pq2 * (r_s_pq2 * r_s_pq2)))^{1/3}))^{3}))
  else
    if (r_s_pq2 < 1.461628127388658E-31) then
      dF = ((((sigma * (sigma * (sigma * (sigma * (sigma * sigma))))) * r_pq) * ((-epsilon) * 24.0)) / ((1 /
           ↪ r_s_pq2) * ((1 / r_s_pq2) * ((1 / r_s_pq2) * (1 / r_s_pq2)))))
    else
      dF = ((-(epsilon * 24.0)) * ((sigma * (sigma * (sigma * (sigma * (sigma * sigma))))) / ((r_s_pq2 * (r_s_pq2 *
           ↪  (r_s_pq2 * r_s_pq2))) / r_pq)))
    end if
  end if
```

Listing 5.3: (Top) Herbie call and output. (Bottom) Translated conditional regime in PPME node annotation.

5.2.2. Discussion

The integration of Herbie into PPME, as presented in section 5.2.1, can be seen as general instructions for connecting external tools. The framework, consisting of action, translators, and an execution model, can easily be adapted to other use cases. Since the implementation is kept in an independent plug-in solution it does not interfere with the base language. Developers are free to use the extension in their simulations.

Future effort could be put into more plug-in extensions for external tooling, hence, taking advantage of existing analysis and optimization programs. As the plug-in integrates directly with the editor, all domain-specific information about the particle-mesh simulation is available to them. This may allow for more sophisticated optimizations not only on single expressions but also across statements, especially over particle loop statements.

Static Analysis Herbie is incorporated as a tool for static analysis of expressions in PPME. Currently, the program is executed upon explicit user request. Technically, performing the analysis can be hooked up with the generator phase in MPS so that it is always executed when generating target source code. We chose manual invocation due to the runtime of the numerical analysis. For arbitrary complex expression Herbie takes 30 seconds up to several minutes for computing an optimized regime (for instance, the analysis of the expression in Figure 5.6 took 43 seconds).

A possible solution to reduce the time PPME spends with analysis is to cache already computed expressions. Optimizations do not need to be recomputed for unchanged expressions. However, modifying an expression should invalidate the annotation, and thus register the expression for the next analysis pass. A database could be used to store a mapping from Herbie test cases to Herbie results with error information and output regime. PPME should first do a lookup for a test case in the database before it invokes the tool. The Herbie GHC plug-in[5] is following this approach.

[5] `https://github.com/mikeizbicki/HerbiePlugin`

Input Error	Output Error
7.1231523	0.08203125

$$\frac{\partial c \to U}{\partial t} = D_u \cdot \nabla^2 c \to U - c \to U \cdot c \to V^2 + F \cdot (1.0 - c \to U)$$

```
<     dU_p = D_u*dU_p - U_p*(V_p**2) + F*(1.0_mk-U_p)
---
>     dU_p = (((constDu * dU_p) - ((U_p * V_p) * V_p)) + ((1.0_mk - U_p) * F))
```

Figure 5.12.: Herbie improvements for **GS-01**.

Input Error	Output Error
2.174412	0.05078125

$$\frac{\partial c \to V}{\partial t} = D_v \cdot \nabla^2 c \to V + c \to U \cdot c \to V^2 - (F + k_{rate}) \cdot c \to V$$

```
<     dV_p = D_v*dV_p + U_p*(V_p**2) - (F+k_rate)*V_p
---
>     dV_p = (((V_p * (U_p * V_p)) + (V_p * (-(kRate + F)))) + (constDv * dV_p))
```

Figure 5.13.: Herbie improvements for **GS-02**.

Accuracy Improvements We investigate the improvements in accuracy for the two case studies, the Lennard-Jones potential (LJ) and the Gray-Scott reaction-diffusion system (GS). We annotated several expressions in each program to be analyzed and optimized by Herbie, and generated the source code with and without taking the optimized regimes into account.

In the Gray-Scott case study (cf. Section 3.3.2) only the two differential equations ($\frac{\partial U}{\partial t}$, **GS-01** and $\frac{\partial V}{\partial t}$, **GS-02**) in the right-hand side definition were tagged for Herbie's analysis. Figure 5.12 depicts the accuracy improvements for the two expressions. Each block shows the input and output error, the original expression, and the difference in the generated source code. Both expressions had a relatively small input error of seven and two bits, respectively. Herbie was able to practically remove inaccuracies by simply expanding and distributing the exponentiation in both cases.

We compared the numerical results for simulations with $t_{end} = 4000$. The final values for the fields U and V differed only slightly in the last four to seven decimal places, which confirms that Herbie's changes do have an influence on the computations. The differences are not notable in the visualization of ParaView as shown in Figure 5.14. However, increasing the runtime may yield a bigger impact of the accuracy improvements.

In the Lennard-Jones example (cf. Section 3.3.1) we annotated four assignment expression. These were the initial assignment to `E_prc` (**LJ-01**), the update of particle positions (**LJ-02**), and the assignments to `dF` and `p->E` in the particle interaction loop (**LJ-03** and **LJ-04**). Figures 5.16 to 5.19 present the results for these four expressions. Overall, Herbie reported a significant improvement for two expressions, and for one expression the bit error could not be reduced at all. In the following, we elaborate these results.

Figure 5.16 shows that the improvement for **LJ-01** is negligibly small. Herbie suggests a complex equivalent expression without actually improving the accuracy. In contrast, the additional square root and exponentiation operations will likely reduce performance.

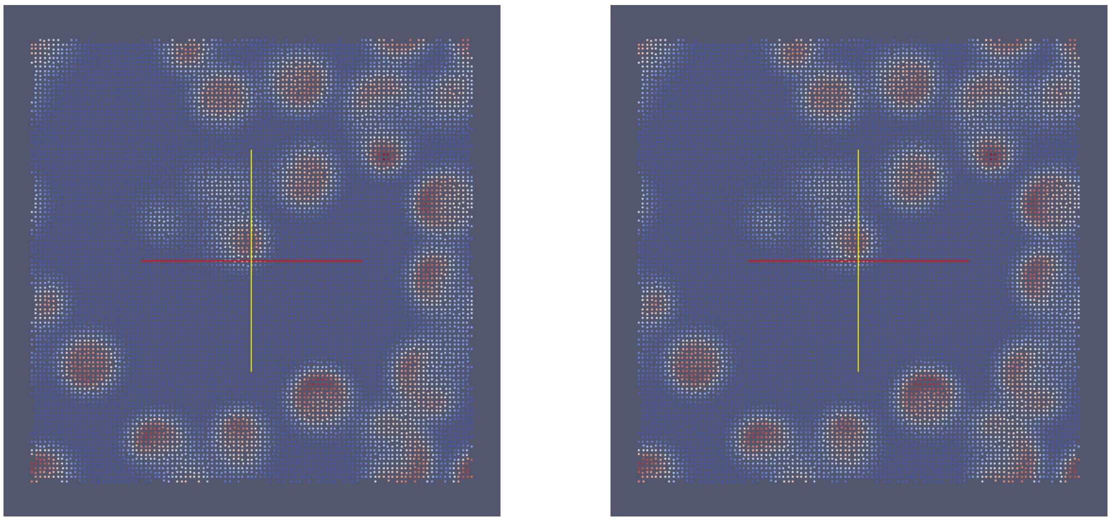


Figure 5.14.: **[LJ]** Visual comparison of the concentration of V at $t = 2000$ generated by ParaView. The small numerical differences are not visible in the representation.

Hence, a developer would likely exclude this expression from the analysis pass and discard the optimization. The resulting expression for **LJ-02** is remarkable as the output error is nearly eliminated by simply expanding the exponentiation of `delta_t` to multiplication. In practice, this kind of optimization might be caught by modern compiler techniques. The assignment to `dF` in **LJ-03** has an reported error of 34 bits, which could be reduced to 15.6 by Herbie. The case distinction branches for very small values of $r_{s_{pq}}$ around zero. Without any further information on value ranges, Herbie removes the term $\sigma^{12}/r^{7}_{s_{pq}}$ completely. Although such optimizations can be reasonable, the developer is advised to verify the changes made by Herbie. Similar results occur for the final optimization of **LJ-04**, where the term $\sigma^{12}/r^{6}_{s_{pq}}$ is ceased. The reported improvement reduces the error from 27.2 to 3.6 bits.

Overall, the results suggest that Herbie was able to automatically improve expressions with numeric instability in the Lennard-Jones case study. However, it has to be noted that the numerical results after including Herbie's modifications differ drastically from the original simulation. The value range for the particle's velocity decreases by an order of magnitude, going from $v_1 \in [-600, 600]$ in the original simulation to $v_1 \in [-40, 40]$ after the modifications. Figure 5.15 shows a side by side comparison of the discrepancy manifested in the visualization output of ParaView. Besides the differences in the data range, it can be seen that the particles are distributed differently. Similarly, the values for the particle's force decrease by two orders of magnitude. So, although Herbie offers hypothetical accuracy improvements, the actual numerical results turn out to be worse than the original implementation.

The emphasized differences can be accounted to Herbie's optimization approach. To Herbie, all input variables are free to range over the real numbers and the corresponding optimization problem takes all those possible values into account. Although the values imported from a control file are unknown at compile-time, they are constant at runtime. Considering this reduces the degree of freedom of the optimization problem, yielding more precise solutions. To cite an example, while Herbie found potential optimizations for **LJ-03** and **LJ-04**, no improvements could be made when replacing the constants ε and σ with their respective values of the default control file. Thus, developers need to be aware of external tool's mechanisms for analysis and program transformations.

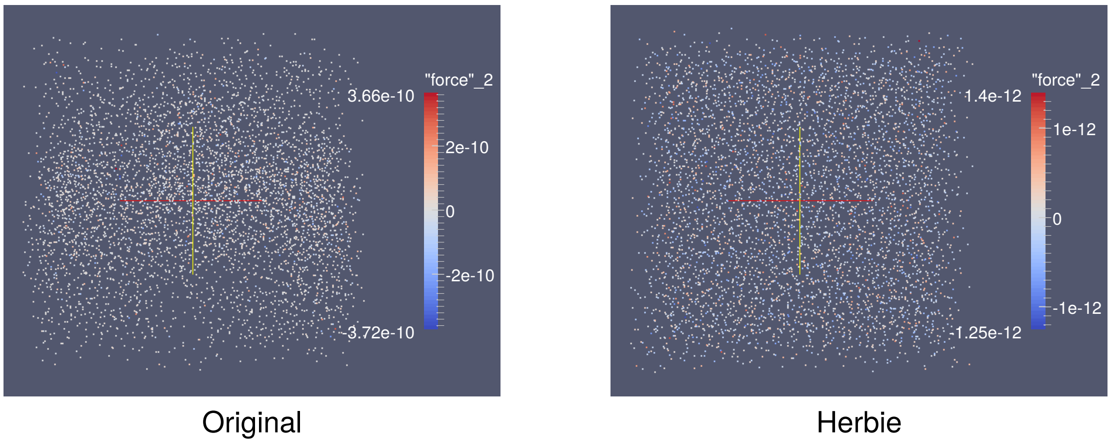


Figure 5.15.: **[LJ]** Visual comparison of the force values f_2 at $t = 0.2$. The value range differs by two orders of magnitude, as can be seen on the scale of the data range.

The theoretical potential for improving numerical stability as identified by Herbie is not reasonable when considering constants and actual variable ranges. From four analyzed expressions only one (**LJ-02**) yielded an actual improvement without affecting the resulting values ($4.16 \mapsto 0.07$). Nevertheless, Herbie's analysis might help developers detecting potential numerical improvements. In order to reduce the occurrence of false optimizations, additional *range annotations* enable a better specification of the free variables. For instance, the equation of **LJ-03** can be annotated with restrictions for $r_{s_{pq}}$, σ, and ϵ.

$$dF = (24.0 \cdot \varepsilon \cdot r_{pq}) \cdot (2.0 \cdot \frac{\sigma^{12}}{r_{s_{pq}}^{7}} - \frac{\sigma^{6}}{r_{s_{pq}}^{4}})$$

where $r_{s_{pq}} > 0, \sigma \in [10^{-2}, 10^{-1}], \varepsilon \in [10^{-14}, 10^{-13}]$

Herbie's documentation describes how this information can be passed to the tool [Her16]. For each variable the distribution it is drawn from can be specified. With the following variable specification in the test case only positive samples are taken for $r_{s_{pq}}$.

```
(r_s (positive default))
```

The range annotations work similarly by restricting the default distribution with an upper and lower bound.

```
(epsilon (< 1.0E-14 default 1.0E-13))
```

Given the additional information, Herbie reports an initial error of five bits for the expression above, and is not able to improve it. This result can ensure developers of the numerical stability of their programs.

In order to make this feature available in PPME, the analysis plug-in provides the concept of range annotations. Any expression can be annotated with an interval (cf. Figure 5.20) appearing as subscript hint $_{\in[\dots]}$ in the editor. The interval specification can be accessed through the inspector view of MPS. We followed the familiar mathematical notation for intervals, assuming the full range of real values per default. This allows to provide additional information on the free variables occurring in a Herbie test case. Figure 5.20 shows the range specifications for ε and $r_{s_{pq}}$. These user annotations are automatically taken into account when generating the test case and running the external analysis. As Herbie can produce better results the more precise its inputs are specified, the additional information supports better results.

Input Error	**Output Error**
2.581556	2.5771723

Input

$$E_{prc} = 4.0 \cdot \varepsilon \cdot (\frac{\sigma}{cutoff + skin})^{12} - (\frac{\sigma}{cutoff + skin})^{6}$$

Output [$c = cutoff, \quad s = skin$]

$$E_{prc} = \left((4 \cdot \varepsilon) \cdot \sqrt{\left(\frac{\sigma}{c+s}\right)^{12}} + (4 \cdot \varepsilon) \cdot \underbrace{\sqrt{\frac{\sigma}{c+s} \cdot (\ldots (\frac{\sigma}{c+s} \cdot \frac{\sigma}{c+s}))}}_{\sqrt{(\frac{\sigma}{c+s})^6}} \right)$$

$$\cdot \left(\left((\sqrt{\left(\frac{\sigma}{c+s}\right)^{12}})^{1/3} \right)^{3} - \underbrace{\sqrt{\frac{\sigma}{c+s} \cdot (\ldots (\frac{\sigma}{c+s} \cdot \frac{\sigma}{c+s}))}}_{\sqrt{(\frac{\sigma}{c+s})^6}} \right)$$

```
<  E_prc = 4.0_mk * epsilon * (((sigma / (cutoff + skin))**12) − ((sigma / (cutoff + skin))**6))
───
>  E_prc = ((((4.0_mk * epsilon) * (sqrt((((sigma / (cutoff + skin))**12))))) + ((4.0_mk * epsilon
     ↪ ) * (sqrt(((sigma / (cutoff + skin)) * ((sigma / (cutoff + skin)) * ((sigma / (cutoff +
     ↪ skin)) * ((sigma / (cutoff + skin)) * ((sigma / (cutoff + skin)) * (sigma / (cutoff +
     ↪ skin)))))))))))) * ((((((sqrt((((sigma / (cutoff + skin))**12))))**1 / 3))**3)) − (sqrt(((
     ↪ sigma / (cutoff + skin)) * ((sigma / (cutoff + skin)) * ((sigma / (cutoff + skin)) * ((
     ↪ sigma / (cutoff + skin)) * ((sigma / (cutoff + skin)) * (sigma / (cutoff + skin)))))))))
     ↪ )
```

Figure 5.16.: Herbie improvements for **LJ-01**.

Input Error	**Output Error**
4.159862	0.0703125

Input

$$p{\rightarrow}pos = p{\rightarrow}pos + p{\rightarrow}v \cdot delta_t + 0.5 \cdot p{\rightarrow}a \cdot delta_t^2$$

Output

$$p{\rightarrow}pos = 1 \cdot (delta_t \cdot p{\rightarrow}a) \cdot (delta_t \cdot 0.5) + p{\rightarrow}v \cdot delta_t + p{\rightarrow}pos$$

```
<  x_p(:) = x_p(:) + v_p(:) * delta_t + 0.5_mk * a_p(:) * (delta_t**2)
───
>  x_p(:) = (1 * (((delta_t * a_p(:)) * (delta_t * 0.5_mk)) + ((v_p(:) * delta_t) + x_p(:))))
```

Figure 5.17.: Herbie improvements for **LJ-02**.

Input Error	**Output Error**
34.03037	15.641862

Input

$$dF = (24.0 \cdot \varepsilon \cdot r_{pq}) \cdot (2.0 \cdot \frac{\sigma^{12}}{r_{s_{pq}}^{7}} - \frac{\sigma^{6}}{r_{s_{pq}}^{4}})$$

Output

$$dF = \begin{cases} (-(\varepsilon \cdot 24)) \cdot \left(\frac{\left(\overbrace{(\sigma \cdot (\ldots (\sigma \cdot \sigma)))}^{\sigma^6} \cdot r_{pq}\right)^{1/3}}{\left(r_{s_{pq}} \cdot (r_{s_{pq}} \cdot (r_{s_{pq}} \cdot r_{s_{pq}}))\right)^{1/3}} \right)^{3} & \text{if } r_{s_{pq}} < -3.4035 \cdot 10^{-94} \\ \frac{(\overbrace{(\sigma \cdot (\ldots (\sigma \cdot \sigma)))}^{\sigma^6} \cdot r_{pq}) \cdot ((-\varepsilon) \cdot 24)}{\frac{1}{r_{s_{pq}}} \cdot (\frac{1}{r_{s_{pq}}} \cdot (\frac{1}{r_{s_{pq}}} \cdot \frac{1}{r_{s_{pq}}}))} & \text{if } r_{s_{pq}} < 1.4616 \cdot 10^{-31} \\ (-(\varepsilon \cdot 24)) \cdot \frac{\overbrace{(\sigma \cdot (\ldots (\sigma \cdot \sigma)))}^{\sigma^6}}{(\frac{r_{s_{pq}} \cdot (r_{s_{pq}} \cdot (r_{s_{pq}} \cdot r_{s_{pq}}))}{r_{pq}})} & \text{otherwise} \end{cases}$$

```
<   dF(:) = (24.0_mk * epsilon * r_pq(:)) * (2.0_mk * ((sigma**12) / (r_s_pq2**7)) - ((sigma**6) /
    ↪  (r_s_pq2**4)))
---
>   if ((r_s_pq2 < -3.4035520833325653E-94)) then
>     dF(:) = ((-(epsilon * 24.0_mk)) * (((((((sigma * (sigma * (sigma * (sigma * (sigma * sigma))
    ↪ ))) * r_pq(:))**1 / 3)) / (((r_s_pq2 * (r_s_pq2 * (r_s_pq2 * r_s_pq2)))**1 / 3)))**3)))
>   else
>     if ((r_s_pq2 < 1.461628127388658E-31)) then
>       dF(:) = ((((sigma * (sigma * (sigma * (sigma * (sigma * sigma))))) * r_pq(:)) * ((-epsilon)
    ↪  * 24.0_mk)) / ((1 / r_s_pq2) * ((1 / r_s_pq2) * ((1 / r_s_pq2) * (1 / r_s_pq2)))))
>     else
>       dF(:) = ((-(epsilon * 24.0_mk)) * ((sigma * (sigma * (sigma * (sigma * (sigma * sigma)))))
    ↪ / ((r_s_pq2 * (r_s_pq2 * (r_s_pq2 * r_s_pq2))) / r_pq(:))))
>     end if
>   end if
```

Figure 5.18.: Herbie improvements for **LJ-03**.

Input Error	**Output Error**
27.181412	3.636401

Input

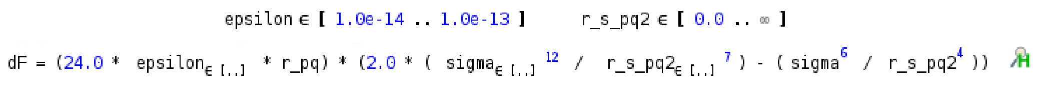
$$p{\to}E = p{\to}E + 4 \cdot \varepsilon \cdot (\frac{\sigma^{12}}{r_{s_{pq}}^{6}} - \frac{\sigma^{6}}{r_{s_{pq}}^{3}}) - E_{prc}$$

Output

$$p{\to}E = \begin{cases} (p{\to}E - E_{prc}) + (-\frac{4}{\varepsilon} \cdot \frac{\sigma \cdot (\sigma \cdot (\sigma \cdot (\sigma \cdot (\sigma \cdot \sigma))))}{r_{s_{pq}}^{3}}) & \text{if } r_{s_{pq}} < -2.511 \cdot 10^{-57} \\ (p{\to}E - E_{prc}) + (-\varepsilon \cdot \frac{\sigma \cdot (\sigma \cdot (\sigma \cdot (\sigma \cdot (\sigma \cdot \sigma))))}{\frac{1}{r_{s_{pq}}^{3}}} \cdot 4) & \text{if } r_{s_{pq}} < 2.237 \cdot 10^{+46} \\ (p{\to}E - E_{prc}) + (-\frac{4}{\varepsilon} \cdot \frac{\sigma \cdot (\sigma \cdot (\sigma \cdot (\sigma \cdot (\sigma \cdot \sigma))))}{r_{s_{pq}}^{3}}) & \text{otherwise} \end{cases}$$

```
<  E_p = E_p + 4 * epsilon * ((sigma**12) / (r_s_pq2**6) - (sigma**6) / (r_s_pq2**3)) - E_prc
---
>   if ((r_s_pq2 < -2.5113679430174213d-57)) then
>     E_p = ((E_p - E_prc) + (-((4 / epsilon) * ((sigma * (sigma * (sigma * (sigma * (sigma * sigma
    ↪ ))))) / ((r_s_pq2**3))))))
>   else
>     if ((r_s_pq2 < 2.2370121346542206d+46)) then
>       E_p = ((E_p - E_prc) + ((((-epsilon) * (sigma * (sigma * (sigma * (sigma * (sigma * sigma))
    ↪ )))) / (1 / ((r_s_pq2**3)))) * 4))
>     else
>       E_p = ((E_p - E_prc) + (-((4 / epsilon) * ((sigma * (sigma * (sigma * (sigma * (sigma *
    ↪ sigma))))) / ((r_s_pq2**3))))))
>     end if
>   end if
```

Figure 5.19.: Herbie improvements for **LJ-04**.



```
                    epsilon ∈ [ 1.0e-14 .. 1.0e-13 ]      r_s_pq2 ∈ [ 0.0 .. ∞ ]
dF = (24.0 *  epsilon∈ [..]  * r_pq) * (2.0 * ( sigma∈ [..] ^12 /  r_s_pq2∈ [..] ^7 ) - ( sigma^6 / r_s_pq2^4 ))
```

Figure 5.20.: Interval specification for variables ε and $r_{s_{pq}}$ in an Herbie-annotated expression. While ε is restricted to a small range, the squared distance of particles is only guaranteed to be positive.

Influence on Runtime Performance Since Herbie modifies expressions and, in some cases, replaces a single assignment with a complex regime containing several conditional branches, decreased runtime performance might be a concern. Therefore, we investigate the influence of these modifications on the simulation runtime for the two case studies, the Gray-Scott reaction-diffusion system and the Lennard-Jones potential. We compared the runtime of the original program for each use case with one containing Herbie's modifications. To get accurate results that are not affected by write operations on the hard drive the simulations were modified so that no output is generated. The performance benchmarks were run on a system with an Intel Core i3-4160 CPU and 16 GB memory. The operating system was Ubuntu with a Linux kernel version of 4.2.0.

We could not find any significant runtime influence for the Gray-Scott use case. The simulation was run over 4000 steps, determined by $t_{start} = 0.0$, $t_{end} = 2000.0$, and $delta_t = 0.5$ (cf. Section B.1). We did a total of 100 measurements per program. Figure 5.21 shows the variation of execution time in box plot notation.

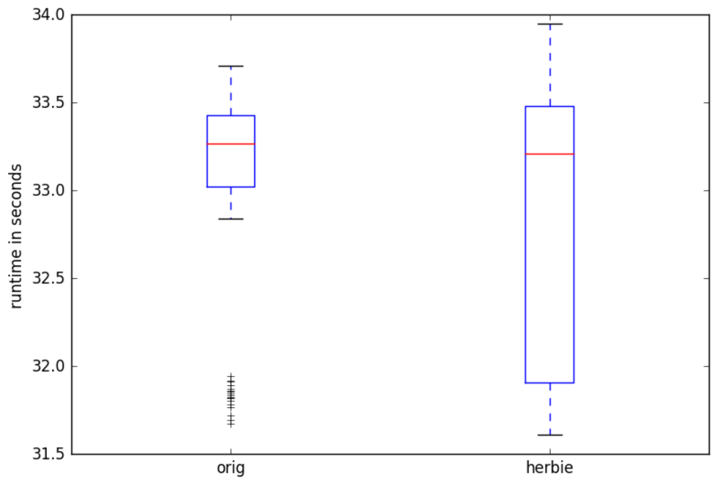


Figure 5.21.: Runtime comparison for the Gray-Scott use case with $t_{end} = 2000$. The median runtime for both simulations is nearly identical, which indicates that Herbie's optimization have no impact on the program's runtime.

One can see that the measured times for the original simulation are less spread than for the optimized version. However, the median in both variants is nearly identical. Furthermore, we observed several outliers with a shorter execution time for the original program, which are approximately at the level of the minimal runtime of the modified simulation.

The measurements indicate that the rearrangement of expressions **GS-01** and **GS-02** does not have any notable influence on the simulation runtime, as could be expected. The median runtime for both programs is nearly identical, and the difference in the spread can be explained by inaccuracies during the benchmark. Still, the question whether a Herbie regime has an influence on the runtime performance remains.

We also compared the runtime of the original PPM client for the Lennard-Jones case study to a version optimized by Herbie. The simulation was executed for $n = 5000$ particles, start time $t_{start} = 0.0$, end time $t_{end} = 0.2$, and time delta $delta_t = 1.0 \cdot 10^{-6}$ (cf. Section B.2). Each variant was run 25 times. The results are summarized in box plot notation in Figure 5.22.

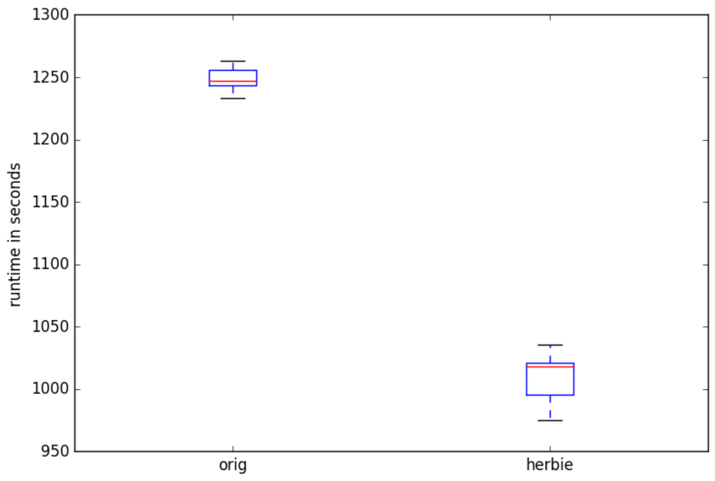


Figure 5.22.: Runtime comparison for the Lennard-Jones case study. The execution of the program modified by Herbie was approximately $20\ \%$ faster than the original implementation.

Surprisingly, the program modified by Herbie runs nearly $20\ \%$ faster than the original implementation. However, this can be attributed to over-simplifications of expressions. As pointed out in Section 5.2.2, the numerical results of the optimized program were less accurate compared to the original PPM client. When taking variable ranges into account, Herbie was not able to find improvements for **LJ-03** and **LJ-04**. Hence, ceasing out the over-complicated rewrite of **LJ-01** and the unchanged expressions **LJ-03** and **LJ-04**, Herbie reported an improvement only for **LJ-02** through a simple restructuring of the expression's terms. The runtime is not impacted by this modification.

The presented plug-in for PPME provides an interface to the main functionalities for Herbie. The analysis of annotated expressions is triggered manually by the developer, where each expression is inspected in every run. To improve the tool's usability, already computed results should be cached in order to avoid unnecessary recomputation. Furthermore, the available interface can be extended to cover more of Herbie's options, e. g., control the transformations which are allowed to perform.

Overall, the integration of external tools for program analysis and numerical optimizations enables a productive use of existing tools. We have presented a general framework for connecting a third party tool, i. e., the Herbie optimizer for floating-point expressions, with PPME by creating an interface for the tool inside the IDE. A simulation can be annotated for optimizations on expression-level by developers. Future extensions can easily integrate new tools with PPME following the same approach.

The presented integration of Herbie aims towards improving the accuracy of floating-point expressions. On the contrary, simulation developers might be interested in sacrificing accuracy for a better performance of numerical operations. The combination of domain-knowledge, user annotations (e. g., interval specifications for variable ranges), and tool integration paves the way for other sophisticated analysis and optimization plug-ins for PPME.

6. Evaluation And Outlook On Future Work

In this final chapter we draw a conclusion on the objectives of this thesis and present an outlook on future work. The general value of PPME to reduce the knowledge gap and simplify development of particle-based simulations using the PPM middleware is discussed. Therefore, the contributions of this thesis are evaluated and put in context for researchers and developers. Additionally, the maturity and suitability of MPS for the implementation of a particle-mesh IDE is questioned. Finally, we propose continuative thoughts and ideas on the further improvement and development of PPME.

In Section 6.1 the contributions made in this thesis are reviewed. In particular, PPME's capability to reduce the knowledge gap and to improve the development process of scientific simulations is discussed. Section 6.2 points out some concerns of MPS in productive use. The importance of model transformations and generators within MPS is addressed in Section 6.3. We argue that Fortran as an additional base language means a significant improvement for adaptability. Finally, in Section 6.4 a roadmap for PPME is presented.

6.1. Review of Contributions

Overall, PPME is an adaptable and extensible workbench for particle-mesh simulations. It aims to simplify the development of scientific simulations through domain-specific abstractions for particle- and particle-mesh based simulations. PPME uses a generative approach to produce PPM client code from a simulation. The initial prototype implementation hinted at the possibilities given by a projectional editor and language workbench technology. Besides syntax highlighting and code completion features the MPS language workbench promises modularity and extensibility for custom languages. This thesis builds upon this prototype. All three contributions [C1] to [C3] address some part of PPME and further improve it. We aimed towards general solutions to demonstrate the flexibility of the system. The first contribution of extending PPME to support an additional case study builds the foundation for the extensions presented in the course of the thesis. The implementation of physical-unit annotations demonstrates an optional language extensions to improve type safety. With the integration of an external tool we demonstrated how to reuse programs for analysis and optimization.

[C1] Improvements to PPME and Case Studies With contribution [C1], the improvement and extension of PPME, we showed the implementation of two case studies, the Gray-Scott reaction-diffusion system and the Lennard-Jones potential, which were both transfered from the implementations inPPML. The different case studies underline PPME's capability to write simulations for both field- and item-based models. The two example simulations cover the main parts of particle-based simulations, e. g., setting up topology and particle list, declaring fields and properties over particles, and defining particle interaction and evolution. PPME avoids many errors by design, e. g., every referenced variable has to be defined, and the code generation process ensures that PPML macros are supplied with the required information extracted from the simulation.

The modularity of PPME allowed to easily adapt concepts to the requirements of the case studies. The different language aspects provided by MPS enabled for specific changes and tweaks to the editor behavior. However, some major refactorings were necessary to fully support both case studies. In addition, several macro concepts corresponding to macros in PPML were introduced to utilize the internal code transformations of MPS. Section 6.3 elaborates this aspect in more detail.

[C2] Types and Units The design and implementation of a formal type system further hardens the error detection of PPME. It prevents a series of common errors at development-time and provides developers with meaningful feedback, e. g., it highlights where an operation is unsupported for its operands or where the types in an assignment expression do not match. We designed the type system with respect to the domain-specific concepts of particle based simulations. MPS allows for a modular implementation of typing rules as it does for language concepts itself. The standard rules for numerical operations on integers or floating-point numbers could be easily extended to cover vectors and particle fields.

One drawback of PPML was that compile errors are reported on the level of the target language, which is Fortran in this case. Due to the applied macro expansions by the PPML compiler, these error message are confusing for a developer working with the PPML front-end. PPME ceases out this disadvantage and instead presents error message on the level of the DSL to the user.

The second part of contribution [C2], the implementation of annotations for physical units, illustrates how optional language extensions can be provided. Units of measure can be annotated on types and expressions by the developer. The additional information enriches the program model and are used to prevent semantic errors due to unit mismatches and conversion errors. The presented unit annotations are solely used within the IDE and do not influence the code generation.

The physical-units extension allows developers to freely define primitive and derived units and use them in their simulations. In particular, unit annotations help to catch semantic errors at development time. As pointed out in Section 4.2.3, the language extension can potentially be expanded to a full dimensional analysis. Furthermore, automatic conversion of quantities with different factors can improve the program's usability.

[C3] Numerical Optimizations The third objective is to investigate numerical optimizations for particle-based simulations. Contribution [C3] addresses this objective by focusing on floating-point expressions. A key question is how to incorporate domain-knowledge in analyses and how to use it to improve the generated simulation programs.

First, rogram equivalence graphs as a representation of semantically equivalent rewrites of a given numerical expression were evaluated. The search space of equivalent expression, emerging from the graph, can be explored to find an expression optimized for some arbitrary property, e. g., improving accuracy. Although the notion of program equivalence graphs is promising to find optimizations on expression-level, we decided against an implementation in PPME as it would have been too comprehensive.

Instead, we considered the integration of existing tools. The second part of [C3] presents a general framework for connecting with external tools, i. e., configuring and executing an external program, and evaluating its output. Conceptually, PPME simulations (or parts thereof) form the input for external analysis tools. We chose Herbie, a tool for improving the accuracy of floating-point expressions, as an example. By leveraging MPS's mechanisms for plug-in integration, Herbie seamlessly integrates with the environment. Parameters for the tool's executable can be set via a native settings dialog in PPME. We enable the annotation of expressions for optimization passes and translate the results back into PPME. This framework for the adaption of external tools can easily be utilized for the integration of other tools.

Finally, we provided an evaluation of the applied optimizations. For both case studies, selected expressions were marked for Herbie analysis. We compared numerical results and performance for the original PPM clients and simulations containing the rewrites proposed by Herbie. Overall, the accuracy of computations in the implemented use cases could not be notably improved, but for the Lennard-Jones potential Herbie oversimplified expressions, yielding less accurate numeric results. We could not measure any significant impact on the simulation runtime for the given case studies.

In order to avoid oversimplification of expression by Herbie, additional value-range annotations were introduced. These annotations allow to further specify the interval of a variable or expression, providing extra information on the domain through user input. Taken into account the range specifications, Herbie was not able to improve the accuracy of the annotated expressions.

6.2. Maturity of MPS

We decided for MPS as base for the implementation of PPME as it does not only provide the typical features of a language workbench, but also allows for advanced representations of language concepts. We utilize this to provide mathematical notations close to the idiom of the domain PPME is designed for. Projects such as mbeddr demonstrate the instantiation of MPS in praxis. The modularity of MPS languages allows for the reuse of common concepts, e. g., multi-line plain text blocks in the projectional editor.

The clean separation of concerns aids the development of specific language aspects. The internal transformation system allows to inspect the full model, and thus enables retrieval of domain-specific information implicitly encoded in the simulations. Projectional editing renders the need for parsers obsolete and allows for complex language structures and automatic transformations.

However, MPS has a steep learning curve, and he amount of different language aspects, aspect specifications, and DSLs easily overcharges new developers. The use of different DSLs throughout the IDE is inconsistent, i. e., similarly looking concepts behave differently, and the documentation lacks detail in some critical sections. Although MPS offers integration with version control systems, serious issues arose in collaborative scenarios. We encountered merge conflicts that could not be resolved using the IDE's internal tools. A manual resolution of merge conflicts is nearly impossible due to the XML format used to store program graphs, containing cryptic node ids and cross-references.

6.3. Base Languages and Generators

MPS promotes the use of internal generators and transformations instead of direct text generation. Transformations allow for more control for rewrites and translations, usually lowering the the language level from domain-specific to general purpose. The goal is to eventually perform all transformations within MPS and only produce a final output for the finished program. From a practical perspective, only one textgen component can be specified for language concepts, but the use of multiple generators is possible.

The current implementation of the transformation and generation pipeline in PPME is shown in Figure 6.1. In a single transformation step the original simulation module is enriched with information which is implicitly available in the model. From this high-level, domain-specific model the textual output against PPML is generated. At this point, the system's boundaries are passed and PPME cannot influence the further processing steps. In particular, the domain-specific knowledge present in the original program is lost for further processing unless previously extracted and exported.

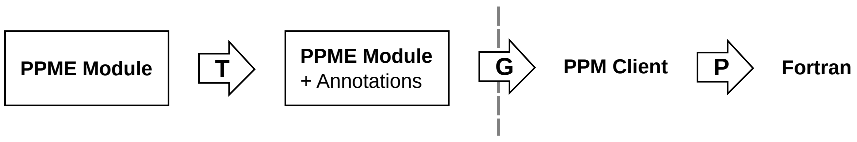


Figure 6.1.: Current transformation and generation pipeline of PPME.

Putting emphasis on more internal transformations to prevent early text generation requires the availability of base languages in MPS. Base languages are often counterparts to general purpose programming languages such as Java. For PPME, a counterpart for Fortran in MPS is required to properly bypass the generation of PPML code and cease out the macro expansion. The logic contained in the macros of PPML is transfered to internal generators.

Generators and transformations enable a smooth adaption to changes in the backend. Source code in different target languages can be produced from a single high-level language by providing transformations to another base language. Hence, for a new C++ implementation of PPM only a set of generators to a C++ base language has to be defined. Notably, a developer would be free to choose the target of the generation. Figure 6.2 depicts the improved transformation pipeline with two possible target languages.

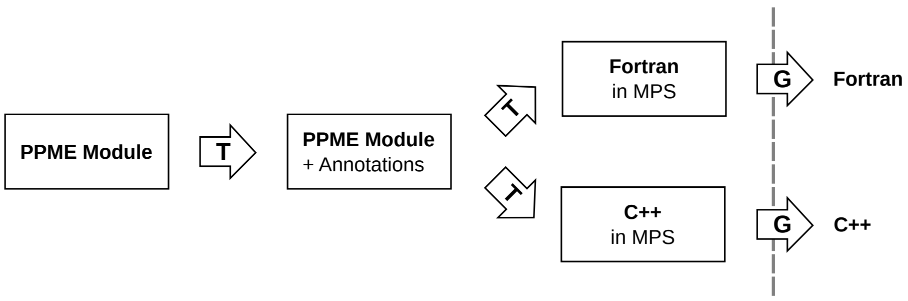


Figure 6.2.: Improvement transformation pipeline with optional target language.

Since the creation of base languages can be cumbersome for complex languages, such as Fortran or C++, an automated approach is desirable. For many common programming languages grammars describing their syntax exist (cf. the collection of grammars for ANTLR[1]). Utilizing these specifications to quickly create new base lan-

[1]`https://github.com/antlr/grammars-v4`

guages might be a viable option if the target language is not available for MPS yet. The *ANTLR-MPS plug-in*[2] provides support to create new MPS languages using ANTLR grammars as a guide. The plug-in could simplify the creation of base languages enormously and thus enable for improvements in PPME's transformation and generation pipeline.

6.4. A Roadmap for PPME

In this final section, we propose a roadmap for PPME. We grouped the upcoming duties in short-, mid-, and long-term goals. The short-term goals address immediate usability improvements and maintenance tasks. Mid-term objectives aim at enhancements in a broader context, i. e., looking at user feedback, new case studies, and invocation of simulations from within the IDE. On long-term perspective, PPME should progress to a stand-alone editor for particle-mesh simulations. Furthermore, the improved generation pipeline would render the use of PPML obsolete.

Short-term Goals The next steps in the development of PPME should address usability improvement such as code completion and automatic program transformations to correct common errors. Tweaking the editor behavior even further should guarantee a flawless development experience.

During the work on this thesis the team around Herbie published a final release (version 1.0.0). In comparison to the development version in use the API for invoking the tool have changed, and a new, standardized input format was adapted. The Herbie plug-in should to be adjusted to these changes. Moreover, the tool execution should be extracted to a background thread, keeping the editor in a productive state even when analyzing numerous expressions. Further speed up can be gained by caching already computed optimizations

Mid-term Goals Continuing from the smaller enhancements, the overall working experience of PPME should be improved. Most of all, the IDE has to be evaluated in practical application. PPME language and editor mechanisms need to be improved based on feedback by developers. New and unforeseen use cases potentially require extension and adaption of the system.

Another important aspect in this context is the ability to compile and execute simulations from the editor. Similar to the invocation of an external tool, the process of compiling a PPM client to Fortran code and executing it could be integrated into PPME. Developers should be able to link to a local installation of PPM and trigger code generation, compilation, and execution with a simple click on a button.

Long-term Goals The long-term goal for PPME is to offer researchers a stand-alone IDE for particle-mesh simulations. MPS allows for packaging a languages and plug-ins in stripped-down variant for this purpose. This toolbox should be enhanced by tools and analysis mechanisms for different use cases. Putting emphasis on internal transformations and base languages, the usage of PPML can be ceased out. Following these steps, PPME can be prepared for future versions of PPM.

[2]`https://plugins.jetbrains.com/plugin/7815`

Appendices

A. Listings

A.1. PPME Generator Scripts

```
mapping script replaceRandoms

script kind : pre-process input model
modifies model : true

(genContext, model, operationContext)->void {
  int cnt = 0;
  // list of variable declarations for random numbers
  nlist<VariableDeclarationStatement> decls = new nlist<VariableDeclarationStatement>;

  // find all expressions containing RNEs
  nlist<Expression> exprs = model.nodes(RandomNumberExpression).select({~rne =>
    node<Expression> rootExpr = rne;
    while (rootExpr.parent.isInstanceOf(Expression)) {
      rootExpr = rootExpr.parent : Expression;
    }
    return rootExpr;
  }).distinct.toList;

  foreach e in exprs {
    // keep track of variables already referenced in expression 'e'
    nlist<VariableDeclarationStatement> used = new nlist<VariableDeclarationStatement>;

    foreach rne in e.descendants<concept = RandomNumberExpression> {
      node<LocalVariableDeclaration> varDecl;
      node<VariableDeclarationStatement> stmnt;

      // check whether a variable can be reused - if not, create a new variable declaration
      if (rne.type.isNull) {
        stmnt = decls.where({~it => it.variableDeclaration.type.isInstanceOf(RealType); }).disjunction(used).first;
        varDecl = <LocalVariableDeclaration(
          name: "rnd_" + cnt,
          type: RealType())>;
      } else {
        stmnt = decls.where({~it => it.variableDeclaration.type == rne.type; }).disjunction(used).first;
        varDecl = <LocalVariableDeclaration(
          name: "rnd_" + cnt,
          type: # rne.type)>;
      }

      // no variable for reuse was found - create a new declaration statement and add it to the list
      if (stmnt.isNull) {
        stmnt = <VariableDeclarationStatement(variableDeclaration: # varDecl)>;
        decls.add(stmnt);
        rne.containing root.descendants<concept = Phase>.first.body.statement.addFirst(stmnt);
        cnt += 1;
      }
      // add the referenced variable to the list of 'used' variables
      used.add(stmnt);

      node<VariableReference> varRef = <VariableReference(variableDeclaration: # stmnt.variableDeclaration)>;
      rne.replace with(varRef);
      e.ancestor<concept = Statement>.add prev-sibling(<CallRandomNumber(var: # varRef)>);
    }
  }
}
```

Listing A.1: `de.ppme.modules.generator.template.main.`
`↪ replaceRandoms`

```
mapping script transformForEachStatements

script kind     : pre-process input model
modifies model  : true

(genContext, model, operationContext)->void {
  foreach particleLoop in model.nodes(ParticleLoopStatment) {
    node<PartLoopsMacro> loop = new node<PartLoopsMacro>();

    nlist<AssignmentExpression> assignments = particleLoop.body.descendants<concept = AssignmentExpression>.toList;
    nlist<Expression> lhs = assignments.left.toList;
    boolean writeX = lhs.findFirst({~it =>
      if (it.isInstanceOf(ArrowExpression) && it : ArrowExpression.operation.isInstanceOf(PositionMemberAccess)) {
        if (it : ArrowExpression.operand.isInstanceOf(VariableReference)) {
          return it : ArrowExpression.operand : VariableReference.variableDeclaration == particleLoop.variable;
        }
      }
      return false;
    }).isNotNull;
    loop.writePos = writeX;

    loop.pset.set(particleLoop.iterable);
    loop.variable.set(particleLoop.variable.copy);

    nlist<ArrowExpression> arrows = particleLoop.body.descendants<concept = ArrowExpression>.distinct.toList;
    nlist<ArrowExpression> particleMemberAccesses = arrows.where({~it => it.operation.isInstanceOf(
         ↪ ParticleMemberAccess); }).toList;

    particleMemberAccesses
      .where({~it => it.operand.isInstanceOf(VariableReference); })
      .where({~it => it.operand : VariableReference.variableDeclaration == particleLoop.variable; })
      .forEach({~it => {
        node<IVariableDeclaration> decl = it.operation : ParticleMemberAccess.decl;
        node<VariableReference> ref = new node<VariableReference>();
        ref.variableDeclaration.set(decl);

        concept switch (decl.type) {
          subconcept of FieldType :
            if (decl.type : FieldType.ndim == 1) {
              if (loop.sca_field.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.
                   ↪ variableDeclaration == ref.variableDeclaration; }).isNull) {
                loop.sca_field.add(ref);
              }
            } else {
              if (loop.vec_field.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.
                   ↪ variableDeclaration == ref.variableDeclaration; }).isNull) {
                loop.vec_field.add(ref);
              }
            }
            skip;
          subconcept of PropertyType :
            if (decl.type : PropertyType.ndim.isInstanceOf(IntegerLiteral) && decl.type : PropertyType.ndim :
                 ↪ IntegerLiteral.value == 1) {
              if (loop.sca_props.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.
                   ↪ variableDeclaration == ref.variableDeclaration; }).isNull) {
                loop.sca_props.add(ref);
              }
            } else {
              if (loop.vec_props.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.
                   ↪ variableDeclaration == ref.variableDeclaration; }).isNull) {
                loop.vec_props.add(ref);
              }
            }
        }
      } });

    loop.body.set(particleLoop.body);
    loop.body.descendants<concept = VariableReference>.where({~it => it.variableDeclaration == particleLoop.
         ↪ variable; }).forEach({~it =>
      node<VariableReference> ref = it.replace with new(VariableReference);
      ref.variableDeclaration.set(loop.variable);
    });

    particleLoop.replace with(loop);
  }
}
```

Listing A.2: `de.ppme.modules.generator.template.main.`
`↪ transformForEachStatements`

```
mapping script populateRHS

script kind    : pre-process input model
modifies model : true

(genContext, model, operationContext)->void {
  foreach ode in model.nodes(ODEStatement) {
    node<RightHandSideMacro> result = new node<RightHandSideMacro>();

    // find out which particle list the ODE is operating on - stop generation when the ambigious
    nlist<Expression> accessedPLists = ode.descendants<concept = ArrowExpression>.operand
      .where({~it => it.type.isInstanceOf(ParticleListType); }).toList;
    if (accessedPLists.select({~it => it.type : ParticleListType.plist; }).distinct.size > 1) {
      error "More than one particle list is accessed in the ODE! Stopping ...";
      return;
    }
    node<Expression> plist = accessedPLists.first;
    result.plist.set(plist.copy);

    // generate a unique name for the right-hand side in the client
    string rhs_name = ode.containing root : Module.getFormattedName();
    rhs_name += "_rhs_" + model.nodes(ODEStatement).indexOf(ode);
    result.name.set(rhs_name);

    // populate list of differential operators
    result.diffops.clear;
    ode.descendants<concept = DifferentialOperator>.forEach({~it => {
        if (result.diffops.where({~elem => it.equals(elem); }).isEmpty) {
          result.diffops.add(it.copy);
        }
      } });

    // create field declarations for the used differential operators
    result.diffops.forEach({~it => {
        node<LocalVariableDeclaration> decl = new node<LocalVariableDeclaration>();
        decl.name.set("d" + it.operand.operation : ParticleListMemberAccess.decl.name);
        decl.type.set(it.operand.getType().copy);
        result.appliedOps.add(decl);
      } });

    info "DiffOps in RHS [" + rhs_name + "]: " + result.diffops.select({~it => it.getDetailedPresentation(); });

    // populate the RHS particle loop
    nlist<Statement> stmnts = ode.body.statement;
    node<VariableDeclaration> loopvar = <LocalVariableDeclaration(
                                          name: "p",
                                          type: ParticleType())>;
    // a particle loop over the previously defined particle list with the loop variable 'p' and an empty statement
          ↪ list
    node<PartLoopsMacro> loop = <PartLoopsMacro(
                                  body: StatementList(),
                                  variable: # loopvar,
                                  pset: # plist)>;

    foreach s in stmnts {
      if (s.isInstanceOf(RightHandSideStatement)) {
        node<RightHandSideStatement> rhsStmnt = s : RightHandSideStatement;
        info "Transforming right-hand side equation [" + rhsStmnt.getDetailedPresentation() + "]";
        node<AssignmentExpression> expr = new node<AssignmentExpression>();

        // the left-hand side of the equation is an access to the diff'op
        // find all occurrences of differential operators on the right hand side that match both the operand and
              ↪ operation of 'X' in 'dX/dt'
        node<IVariableDeclaration> diffField = rhsStmnt.argument.operation : ParticleListMemberAccess.decl;
        nlist<DifferentialOperator> diffops =
          result.diffops.distinct
            .where({~it => it.operand.operand.isInstanceOf(VariableReference) && it.operand.operand :
                  ↪ VariableReference.variableDeclaration == plist : VariableReference.variableDeclaration; })
            .where({~it => it.operand.operation.isInstanceOf(ParticleListMemberAccess) && it.operand.operation :
                  ↪ ParticleListMemberAccess.decl == diffField; }).toList;

        if (diffops.size > 1) {
          error "Found more than one potential differential operator! Stopping ...";
          return;
        } else if (diffops.isEmpty) {
          error "No matching differential operator found! Stopping ...";
          return;
        } else {
          info "Selected DiffOp: " + diffops.first + "<" + diffops.first.operand.getDetailedPresentation() + ">";
        }

        node<DifferentialOperator> diffop = diffops.first.copy;
        diffop.operand.set(<ArrowExpression(
          operand: VariableReference(variableDeclaration: # loop.variable),
          operation: ParticleMemberAccess(decl: # rhsStmnt.argument.operation : ParticleListMemberAccess.decl))>);

        node<Expression> left = diffop;
        node<Expression> right = rhsStmnt.rhs;

        // transform ParticleListMemberAccesses to ParticleMemberAccesses in the loop
        right.descendants<concept = ArrowExpression>.where({~it => it.operation.isInstanceOf(
              ↪ ParticleListMemberAccess); }).forEach({~it =>
          node<> t;
          if (it.operation : ParticleListMemberAccess.decl.isInstanceOf(VariableDeclaration)) {
            t = it.operation : ParticleListMemberAccess.decl : VariableDeclaration.getType() ;
          } else {
            t = it.operation : ParticleListMemberAccess.decl.type : Type;
          }

        node<ArrowExpression> foo = it.replace with new(ArrowExpression);
        foo.operand.set(<VariableReference(variableDeclaration: # loop.variable)>);
        foo.operation.set(<ParticleMemberAccess(decl: # it.operation : ParticleListMemberAccess.decl)>);
      });

      expr.left.set(left);
      expr.right.set(right);

      // populate fields and properties in use
      expr.right.descendants<concept = ParticleMemberAccess>.where({~it => !it.getOperand().parent.isInstanceOf(
           ↪ DifferentialOperator); }).where({~it => it.getOperand().isInstanceOf(VariableReference) && it.
           ↪ getOperand() : VariableReference.variableDeclaration == loop.variable; }).forEach({~it =>
        node<IVariableDeclaration> decl = it.decl;
        node<Type> t = (decl.isInstanceOf(VariableDeclaration)) ? decl : VariableDeclaration.getType() : decl.
             ↪ type : Type;
        info "found particle member access ... [" + it.getDetailedPresentation() + "::" + t.
             ↪ getDetailedPresentation() + "]";
        node<VariableReference> ref = new node<VariableReference>();
        ref.variableDeclaration.set(decl);

        concept switch (decl : VariableDeclaration.type) {
          subconcept of FieldType :
            if (decl.type : FieldType.ndim == 1) {
              if (loop.sca_field.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.
                   ↪ variableDeclaration == ref.variableDeclaration; }).isNull) {
                loop.sca_field.add(ref);
              }
            } else {
              if (loop.vec_field.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.
                   ↪ variableDeclaration == ref.variableDeclaration; }).isNull) {
                loop.vec_field.add(ref);
              }
            }
            skip;
          subconcept of PropertyType :
            if (decl.type : PropertyType.ndim.isInstanceOf(IntegerLiteral) && decl.type : PropertyType.ndim :
                 ↪ IntegerLiteral.value == 1) {
              if (loop.sca_props.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.
                   ↪ variableDeclaration == ref.variableDeclaration; }).isNull) {
                loop.sca_props.add(ref);
              }
            } else {
              if (loop.vec_props.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.
                   ↪ variableDeclaration == ref.variableDeclaration; }).isNull) {
                loop.vec_props.add(ref);
              }
            }
        }
      });

      loop.body.statement.add(<ExpressionStatement(expression: # expr)>);
    } else {
      s.descendants<concept = RightHandSideStatement>.forEach({~eq =>
        info "Transforming right-hand side equation [" + eq.getDetailedPresentation() + "]";
        node<AssignmentExpression> expr = new node<AssignmentExpression>();
        node<IVariableDeclaration> diffField = eq.argument.operation : ParticleListMemberAccess.decl;
        nlist<DifferentialOperator> diffops = result.diffops.distinct.distinct.distinct.distinct.where({~it => it
             ↪ .operand.operand.isInstanceOf(VariableReference) && it.operand.operand : VariableReference.
             ↪ variableDeclaration == plist : VariableReference.variableDeclaration; }).where({~it => it.
             ↪ operand.operation.isInstanceOf(ParticleListMemberAccess) && it.operand.operation :
             ↪ ParticleListMemberAccess.decl == diffField; }).toList;

        if (diffops.size > 1) {
          error "Found more than one potential differential operator! Stopping ...";
          return;
        } else if (diffops.isEmpty) {
          error "No matching differential operator found! Stopping ...";
          return;
        } else {
          info "Selected DiffOp: " + diffops.first + "<" + diffops.first.operand.getDetailedPresentation() + ">";
        }

        node<DifferentialOperator> diffop = diffops.first.copy;
        diffop.operand.set(<ArrowExpression(
          operand: VariableReference(variableDeclaration: # loop.variable),
          operation: ParticleMemberAccess(decl: # eq.argument.operation : ParticleListMemberAccess.decl))>);

        node<Expression> left = diffop;
        node<Expression> right = eq.rhs;

        // transform ParticleListMemberAccesses to ParticleMemberAccesses in the loop
        right.descendants<concept = ArrowExpression>.where({~it => it.operation.isInstanceOf(
             ↪ ParticleListMemberAccess); }).forEach({~it =>
          node<> t = (it.operation : ParticleListMemberAccess.decl.isInstanceOf(VariableDeclaration)) ? it.
               ↪ operation : ParticleListMemberAccess.decl : VariableDeclaration.getType() : it.operation :
               ↪ ParticleListMemberAccess.decl.type : Type;
          info "found particle list member access: " + it.getDetailedPresentation() + "::" + t.
               ↪ getDetailedPresentation();

          node<ArrowExpression> foo = it.replace with new(ArrowExpression);
          foo.operand.set(<VariableReference(variableDeclaration: # loop.variable)>);
          foo.operation.set(<ParticleMemberAccess(decl: # it.operation : ParticleListMemberAccess.decl)>);
          info "replacing PLMA with PMA ... [" + it.getDetailedPresentation() + "::" + it.operation.type + " ==>
               ↪ " + foo.getDetailedPresentation() + "::" + foo.type + "]";
        });

        expr.left.set(left);
        expr.right.set(right);

```

```
          // populate fields and properties in use
          expr.right.descendants<concept = ParticleMemberAccess>.where({~it => !it.getOperand().parent.isInstanceOf(DifferentialOperator); }).where({~it => it.getOperand().isInstanceOf(VariableReference) && it.getOperand() : VariableReference.variableDeclaration == loop.variable; }).forEach({~it =>

            node<IVariableDeclaration> decl = it.decl;
            node<Type> t = (decl.isInstanceOf(VariableDeclaration)) ? decl : VariableDeclaration.getType() : decl.type : Type;
            info "found particle member access ... [" + it.getDetailedPresentation() + "::" + t.getDetailedPresentation() + "]";

            node<VariableReference> ref = new node<VariableReference>();
            ref.variableDeclaration.set(decl);

            concept switch (decl : VariableDeclaration.type) {
              subconcept of FieldType :
                if (decl.type : FieldType.ndim == 1) {
                  if (loop.sca_field.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.variableDeclaration == ref.variableDeclaration; }).isNull) {
                    loop.sca_field.add(ref);
                  }
                } else {
                  if (loop.vec_field.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.variableDeclaration == ref.variableDeclaration; }).isNull) {
                    loop.vec_field.add(ref);
                  }
                }
                skip;
              subconcept of PropertyType :
                if (decl.type : PropertyType.ndim.isInstanceOf(IntegerLiteral) && decl.type : PropertyType.ndim : IntegerLiteral.value == 1) {
                  if (loop.sca_props.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.variableDeclaration == ref.variableDeclaration; }).isNull) {
                    loop.sca_props.add(ref);
                  }
                } else {
                  if (loop.vec_props.findFirst({~it => it.isInstanceOf(VariableReference) && it : VariableReference.variableDeclaration == ref.variableDeclaration; }).isNull) {
                    loop.vec_props.add(ref);
                  }
                }
            }
          });
          eq.replace with(<ExpressionStatement(expression: # expr)>);
        });
        loop.body.statement.add(s.copy);
      }
    }
    // the fields and properties accessed in the loop are the rhs_vars
    result.vars.clear;
    result.vars.addAll(loop.sca_field.select({~it => it.copy; }));
    result.vars.addAll(loop.vec_field.select({~it => it.copy; }));
    result.vars.addAll(loop.sca_props.select({~it => it.copy; }));
    result.vars.addAll(loop.vec_props.select({~it => it.copy; }));

    result.appliedOps.forEach({~it =>
      info "adding applied operator field to particle loop ... ";
      node<VariableReference> ref = new node<VariableReference>();
      ref.variableDeclaration.set(it);
      node<Type> t = (it.isInstanceOf(VariableDeclaration) ? it : VariableDeclaration.getType() : it.type as Type);
      concept switch (t) {
        subconcept of FieldType :
          if (t : FieldType.ndim == 1) {
            info "-- adding scalar field ... [" + t.getDetailedPresentation() + "]";
            loop.sca_field.add(ref);
          } else {
            info "-- adding vector field ... [" + t.getDetailedPresentation() + "]";
            loop.vec_field.add(ref);
          }
      }
    });
    result.loop.set(loop);
    ode.containing root : Module._rhs.add(result);
  }
}
```

Listing A.3: de.ppme.modules.generator.template.main.populateRHS

```
mapping script HerbieOptimizations

script kind : pre-process input model
modifies model : true

(genContext, model, operationContext)->void {
  foreach expr in model.nodes(AssignmentExpression).where({~it => it.@herbie != null; }) {
    if (expr.@herbie.replacement.isNotNull) {
      node<Statement> parentStmnt = expr.ancestor<concept = ExpressionStatement>;
      if (parentStmnt.isNotNull) {
        info "[Herbie Generator] replacing expression with optimized regime ...";
        parentStmnt.replace with(expr.@herbie.replacement);
      }
    }
  }

  foreach rhss in model.nodes(RightHandSideStatement).where({~rhs => rhs.rhs.@herbie.isNotNull; }) {
    node<Expression> rhsExpr = rhss.rhs;
    if (rhsExpr.@herbie.replacement.isNotNull) {
      info "[Herbie Generator] replacing right-hand side statement with optimized regime ...";
      rhss.replace with(rhsExpr.@herbie.replacement.copy);
    }
  }
}
```

Listing A.4: `de.ppme.analysis.generator.template.main.`
↪ `HerbieOptimizations`

A.2. Physical-Unit Conversion

```
/**
  taken from: https://github.com/fisakov/mps-example-physunits
  original license: Apache License, Version 2.0 (http://www.apache.org/licenses/LICENSE-2.0)
 */

public class PhysicalUnitConversion {


public static nlist<PhysicalUnitRef> expand(nlist<PhysicalUnitRef> spec) {
  list<[node<IPhysicalUnit>, int]> expanded = new arraylist<[node<IPhysicalUnit>, int]>(copy: spec.select({~unitRef => [unitRef.decl, unitRef.getExponent()]; }));

  [node<IPhysicalUnit>, int] pair;
  do {
    pair = expanded.findFirst({~sp => sp[0] as PhysicalUnit.spec.component.isNotEmpty; });
    if (pair != null) {
      expanded.remove(pair);
      int exponent = pair[1];
      expanded.addAll(pair[0] as PhysicalUnit.spec.component.select({~unitRef => [unitRef.decl, exponent * unitRef.getExponent()]; }));
    }
  } while (pair != null);

  sequence<[node<IPhysicalUnit>, int]> expandedSorted = expanded.sortBy({~it => it[0].name; }, asc);
  sequence<string> names = expandedSorted.select({~it => it[0].name; }).distinct;
  list<[node<IPhysicalUnit>, int]> expandedFlatten = names.select({~name => expandedSorted.selectMany({~it =>
      if (it[0].name :eq: name) { yield it; }
    }).reduceLeft({~a,~b => [a[0], a[1] + b[1]]; }); }).toList;
  expandedFlatten.removeWhere({~it => it[1] == 0; });

  expandedFlatten.select({~it => it[1] != 1 ? <PhysicalUnitRef(
      decl: # it[0],
      exponent: Exponent(value: it[1]))> : <PhysicalUnitRef(
      decl: # it[0],
      exponent: null)>; }).toList;
}


public static nlist<PhysicalUnitRef> simplify(nlist<PhysicalUnitRef> spec, nlist<PhysicalUnitDeclarations> decls) {
  list<[node<IPhysicalUnit>, int]> simplified = new arraylist<[node<IPhysicalUnit>, int]>(copy: spec.selectMany({~ur => PhysicalUnitConversion.demultiplex([ur.decl, ur.getExponent()]); }));

  sequence<node<PhysicalUnit>> allUnits = decls.selectMany({~decl => decl.units; });
  sequence<node<PhysicalUnit>> sortedUnits = allUnits.sortBy({~u => u.spec.component.select({~c => Math.abs(c.getExponent()); }).reduceLeft({~a,~b => a + b; }); }, desc);

  boolean done;
  do {
    done = true;
    for (node<PhysicalUnit> cunit : sortedUnits) {
      if (cunit.spec.component.isEmpty) { continue; }

      sequence<[node<IPhysicalUnit>, int]> cspec = cunit.spec.component.selectMany({~ur => PhysicalUnitConversion.demultiplex([ur.decl, ur.getExponent()]); });
      sequence<[node<IPhysicalUnit>, int]> cspecrecip = cunit.spec.component.selectMany({~ur => PhysicalUnitConversion.demultiplex([ur.decl, -ur.getExponent()]); });

      if (simplified.containsAll(cspec)) {
        simplified.removeAll(cspec);
        simplified.add([cunit, 1]);
      } else if (simplified.containsAll(cspecrecip)) {
        simplified.removeAll(cspecrecip);
        simplified.add([cunit, -1]);
      } else {
        continue;
      }

      // start next iteration
      done = false;
      break;
    }
  } while (!done);

  sequence<[node<IPhysicalUnit>, int]> simplifiedSorted = simplified.sortBy({~it => it[0].name; }, asc);
  sequence<string> names = simplifiedSorted.select({~it => it[0].name; }).distinct;

  list<[node<IPhysicalUnit>, int]> simplifiedFlatten = names.select({~name => simplifiedSorted.selectMany({~it =>
      if (it[0].name :eq: name) { yield it; }
    }).reduceLeft({~a,~b => [a[0], a[1] + b[1]]; }); }).toList;
  simplifiedFlatten.removeWhere({~it => it[1] == 0; });

  simplifiedFlatten.select({~it => it[1] != 1 ? <PhysicalUnitRef(
      decl: # it[0],
      exponent: Exponent(value: it[1]))> : <PhysicalUnitRef(
      decl: # it[0],
      exponent: null)>; }).toList;
}


  public static boolean matching(nlist<PhysicalUnitRef> specA, nlist<PhysicalUnitRef> specB) {
    list<[node<IPhysicalUnit>, int]> unwrappedA = new arraylist<[node<IPhysicalUnit>, int]>(copy: specA.select({~unitRef => [unitRef.decl, unitRef.getExponent()]; }));
    list<[node<IPhysicalUnit>, int]> unwrappedB = new arraylist<[node<IPhysicalUnit>, int]>(copy: specB.select({~unitRef => [unitRef.decl, unitRef.getExponent()]; }));
    unwrappedA.disjunction(unwrappedB).isEmpty;
  }


  public static boolean matching(nlist<PhysicalUnitRef> specA, nlist<PhysicalUnitRef> specB, map<node<MetaPhysicalUnit>, node<PhysicalUnitRef>> unifier) {
    list<[node<IPhysicalUnit>, int]> unwrappedA = new arraylist<[node<IPhysicalUnit>, int]>(copy: specA.select({~unitRef => [unitRef.decl, unitRef.getExponent()]; }));
    list<[node<IPhysicalUnit>, int]> unwrappedB = new arraylist<[node<IPhysicalUnit>, int]>(copy: specB.select({~unitRef => [unitRef.decl, unitRef.getExponent()]; }));

    unwrappedA.removeAll(unwrappedB);
    unwrappedB.removeAll(unwrappedA);

    boolean unified = true;

    foreach a in unwrappedA, b in unwrappedB {
      if (a[0].isInstanceOf(MetaPhysicalUnit) && b[0].isInstanceOf(PhysicalUnit)) {
        unified = substituteMetaUnit(a, b, unifier);
      } else if (a[0].isInstanceOf(PhysicalUnit) && b[0].isInstanceOf(MetaPhysicalUnit)) {
        unified = substituteMetaUnit(b, a, unifier);
      } else {
        error "Matching supports only meta unit type and unit type, but was A=" + a + ", B=" + b;
        unified = false;
      }
    }

    return unified;
  }


  public static nlist<PhysicalUnitRef> recip(nlist<PhysicalUnitRef> spec) {
    return spec.select({~it => it.getExponent() != -1 ? <PhysicalUnitRef(
        decl: # it.decl,
        exponent: Exponent(value: -it.getExponent()))> : <PhysicalUnitRef(
        decl: # it.decl,
        exponent: null)>; }).toList;
  }


  private static boolean substituteMetaUnit([node<IPhysicalUnit>, int] metaUnit, [node<IPhysicalUnit>, int] physUnit, map<node<MetaPhysicalUnit>, node<PhysicalUnitRef>> unifier) {
    int exp;

    if (Math.abs(metaUnit[1]) <= Math.abs(physUnit[1]) && physUnit[1] % metaUnit[1] == 0) {
      exp = physUnit[1] / metaUnit[1];
    } else {
      error "Exponent less than 1 by abs value is not supported, was given meta=" + metaUnit + ", unit=" + physUnit ;
      exp = 0;
    }
    if (unifier != null) {
      if (exp != 1) {
        unifier[metaUnit[0] : MetaPhysicalUnit] = <PhysicalUnitRef(
          decl: # physUnit[0],
          exponent: Exponent(value: exp))>;
      } else {
        unifier[metaUnit[0] : MetaPhysicalUnit] = <PhysicalUnitRef(
          decl: # physUnit[0],
          exponent: null)>;
      }
    }
    return exp != 0;
  }


  private static sequence<[node<IPhysicalUnit>, int]> demultiplex([node<IPhysicalUnit>, int] input) {
    int exp = input[1];
    if (exp == 0) { throw new IllegalArgumentException("null exponent"); }
    if (exp == 1) {
      return new singleton<[node<IPhysicalUnit>, int]>(input);
    } else {
      list<[node<IPhysicalUnit>, int]> res = new arraylist<[node<IPhysicalUnit>, int]>;
      int sign = exp < 0 ? -1 : 1;
      for (int i = 0; i < Math.abs(exp); i++) {
        res.add([input[0], sign]);
      }
      return res;
    }
  }
}
```

Listing A.5: de.ppme.physunits/PhysicalUnitConversions.java

B. Benchmarks

B.1. Gray-Scott Case Study

```
  k_rate = 0.05100000000000
 F_param = 0.01500000000000
Du_param = 2E-5
Dv_param = 1E-5
```

⋮

```
            min_phys =   0.000000000000000E+000,   0.000000000000000E+000
            max_phys =    1.00000000000000,    1.00000000000000
domain_decomposition = 7
processor_assignment = 1
          ghost_size = 0.05000000000000
               Npart = 10000
```

⋮

```
ODEscheme = 4
```

⋮

```
start_time = 0.0000000000000000
time_step = 0.50000000000000000
stop_time = 500.00000000000000
```

Listing B.1: `benchmarks/gs/Ctrl` — Control file used for benchmarking the Gray-Scott case study.

```
#!/usr/bin/env python3

import sys
from sys import argv
import subprocess
import os.path
import time
import json

MPICommand = ’mpirun␣-np␣4␣{}’

executables = json.loads(argv[1])
nSamples = int(argv[2])

if not os.path.isfile("Ctrl"):
    print("Error:␣no␣control␣file␣found!")
    sys.exit(1)

ctrl = {}
with open("./Ctrl") as ctrlFile:
    for line in ctrlFile:
        key, value = line.partition("=")[::2]
        ctrl[key.strip()] = value.strip()

root = os.getcwd()

for n in [500, 1000, 2000, 4000]:
    for client, exe in executables.items():
        os.chdir(root)

        test_id = "{}_{}".format(client, n)

        if not os.path.isdir(test_id):
            os.mkdir(test_id)

        ctrl["stop_time"] = n
        with open("{}/Ctrl".format(test_id), ’w’) as f:
            for k, v in ctrl.items():
                f.write("{}␣=␣{}\n".format(k, v))

        e = os.path.join(os.path.abspath(os.getcwd()), exe)
        os.chdir(test_id)

        results = []

        for i in range(nSamples):
            print("Running␣test␣id={},␣i={}".format(test_id, i))
            ts = time.time()
            subprocess.call(MPICommand.format(e), shell=True, stdout=subprocess.DEVNULL)
            te = time.time()
            results.append(te - ts)

        reportFile = os.path.join(os.path.abspath(os.getcwd()), "report")
        with open(reportFile, ’w’) as f:
            f.write(MPICommand.format(e) + "\n")
            for r in results:
                f.write("{}\n".format(r))
```

Listing B.2: `benchmarks/gs/benchmark.py` — Benchmark script for the Gray-Scott case study.

```
client grayscott
  integer, dimension(6) :: bcdef = ppm_param_bcdef_periodic
  integer, dimension(2) :: seed
  real(ppm_kind_double), dimension(:,:), pointer :: displace
  real(ppm_kind_double) :: noise = 0.0_mk
  integer               :: istage = 1
  integer               :: interval

  add_arg(k_rate,<#real(mk)#>,1.0_mk,0.0_mk,'k_rate','Reaction␣rate')
  add_arg(F,<#real(mk)#>,1.0_mk,0.0_mk,'F_param','Reaction␣parameter␣F')
  add_arg(D_u,<#real(mk)#>,1.0_mk,0.0_mk,'Du_param','Diffusion␣constant␣of␣U')
  add_arg(D_v,<#real(mk)#>,1.0_mk,0.0_mk,'Dv_param','Diffusion␣constant␣of␣V')

  ppm_init()

  U = create_field(1, "U")
  V = create_field(1, "V")

  topo = create_topology(bcdef)

  c = create_particles(topo)
  allocate(displace(ppm_dim,c%Npart))
  call random_number(displace)
  displace = (displace - 0.5_mk) * c%h_avg * 0.15_mk
  call c%move(displace, info)
  call c%apply_bc(info)
  !call c%set_cutoff(4._mk * c%h_avg, info)

  global_mapping(c, topo)

  discretize(U,c)
  discretize(V,c)

  ghost_mapping(c)

  foreach p in particles(c) with positions(x) sca_fields(U,V)
    U_p = 1.0_mk
    V_p = 0.0_mk
    if (((x_p(1) - 0.5_mk)**2 + (x_p(2) - 0.5_mk)**2) .lt. 0.01) then
      call random_number(noise)
      U_p = 0.5_mk  + 0.01_mk*noise
      call random_number(noise)
      V_p = 0.25_mk + 0.01_mk*noise
    end if
  end foreach

  n = create_neighlist(c,cutoff=<#4._mk * c%h_avg#>)

  if (ppm_dim .eq. 2) then
    Lap = define_op(2, [2,0, 0,2], [1.0_mk, 1.0_mk], "Laplacian")
  else
    Lap = define_op(3, [2,0,0, 0,2,0, 0,0,2], [1.0_mk, 1.0_mk, 1.0_mk], "Laplacian")
  end if

  L = discretize_op(Lap, c, ppm_param_op_dcpse,[order=>2,c=>1.0_mk])

  o, nstages = create_ode([U,V], grayscott_rhs, [U=>c,V], rk4)
  interval = 1
  !print([U=>c, V=>c],1 )
  t = timeloop()
    do istage=1,nstages
      ghost_mapping(c)
      ode_step(o, t, time_step, istage)
    end do
    !print([U=>c, V=>c],interval )
  end timeloop

  !print([U=>c, V=>c],1 )
  ppm_finalize()
end client

rhs grayscott_rhs(U=>parts,V)
  get_fields(dU,dV)

  dU = apply_op(L, U)
  dV = apply_op(L, V)


  foreach p in particles(parts) with sca_fields(U,V,dU,dV)
    dU_p = D_u*dU_p - U_p*(V_p**2) + F*(1.0_mk-U_p)
    dV_p = D_v*dV_p + U_p*(V_p**2) - (F+k_rate)*V_p
  end foreach

end rhs
```

Listing B.3: `benchmarks/gs/orig_4000/gs.ppm` — Original PPM client code for the Gray-Scott case study as used in the benchmark.

```
client grayscott__single_phase_
  add_arg(constDu, <#real(mk)#>, default=1.0_mk, min=0.0_mk, ctrl_name='constDu', help_txt='diffusion␣constant␣
        ↪ of␣U')
  add_arg(F, <#real(mk)#>, default=1.0_mk, min=0.0_mk, ctrl_name='F', help_txt='reaction␣parameter␣F')
  add_arg(kRate, <#real(mk)#>, default=1.0_mk, min=0.0_mk, ctrl_name='kRate', help_txt='reaction␣rate')
  add_arg(constDv, <#real(mk)#>, default=1.0_mk, min=0.0_mk, ctrl_name='constDv', help_txt='diffusion␣constant␣
        ↪ of␣V')

  real(ppm_kind_double) :: rnd_2
  real(ppm_kind_double) :: rnd_1
  real(ppm_kind_double), dimension(:,:), pointer :: rnd_0
  integer, dimension(6) :: bcdef = ppm_param_bcdef_periodic
  real(ppm_kind_double), dimension(:,:), pointer :: displacement
  integer :: istage

  ppm_init()

  Lap = define_op(2, [2, 0, 0, 2], [1.0_mk, 1.0_mk], "Laplacian")

! begin phase all-in-one
  topo = create_topology(bcdef)

  ! -- creating particle set --------------------
  c = create_particles(topo)

  allocate(rnd_0(ppm_dim,c%Npart))
  call random_number(rnd_0)
  allocate(displacement(ppm_dim, c%Npart))
  displacement = (rnd_0 - 0.5_mk) * c%h_avg * 0.15_mk
  call c%move(displacement, info)

  call c%apply_bc(info)
  global_mapping(c, topo)

  U = create_field(1, "U")
  V = create_field(1, "V")
  discretize(U, c)
  discretize(V, c)

  ghost_mapping(c)
  ! -- end: creating particle set ----------------

  foreach p in particles(c) with positions(x) sca_fields(U, V)
    U_p = 1.0_mk
    V_p = 0.0_mk
    if (((x_p(1) - 0.5_mk)**2) + ((x_p(2) - 0.5_mk)**2) < 0.01_mk) then
      call random_number(rnd_1)
      U_p = 0.5_mk + 0.01_mk * rnd_1
      call random_number(rnd_2)
      V_p = 0.25_mk + 0.01_mk * rnd_2
    end if
  end foreach

  n = create_neighlist(c, cutoff=<#4.0_mk * c%h_avg#>)



  L = discretize_op(Lap, c, ppm_param_op_dcpse, [order=>2, c=>1.0_mk])
  o_0, nstages_0 = create_ode([U, V], grayscott__single_phase__rhs_0, [U=>c, V], rk4)
  !print([U=>c, V=>c], 1)
  t = timeloop()
    do istage=1, nstages_0
      ghost_mapping(c)
      ode_step(o_0, t, time_step, istage)
    end do
    !print([U=>c, V=>c], 1)
  end timeloop
  !print([U=>c, V=>c], 1)

! end phase all-in-one

  ppm_finalize()
end client

rhs grayscott__single_phase__rhs_0(U=>c, V)
  get_fields(dU, dV)

  dU = apply_op(L, U)
  dV = apply_op(L, V)

  foreach p in particles(c) with positions(x) sca_fields(U, V, dU, dV)
    dU_p = (((constDu * dU_p) - ((U_p * V_p) * V_p)) + ((1.0_mk - U_p) * F))
    dV_p = (((V_p * (U_p * V_p)) + (V_p * (-(kRate + F)))) + (constDv * dV_p))
  end foreach

end rhs
```

Listing B.4: `benchmarks/lj/herbie_4000/gs.ppm` — Generated PPM client code for the Gray-Scott case study as used in the benchmark.

B.2. Lennard-Jones Case Study

```
#-- CTRL ---------------------------------------
# Generated from 'default'
#-----------------------------------------------

mass = 6.69E-18
epsilon = 1.65677856E-13
sigma = 3.605E-2
delta_t = 1.0E-6
min_phys = 0.0, 0.0, 0.0
max_phy = 1.0, 1.0, 1.0
domain_decomposition = 7
processor_assignment = 1
ghost_size = 0.1
Npart = 5000
start_time = 0.0
stop_time = 0.2
```

Listing B.5: `benchmarks/lj/Ctrl` — Control file used for benchmarking the Lennard-Jones case study.

```python
#!/usr/bin/env python3

import sys
from sys import argv
import subprocess
import os.path
import time
import json

MPICommand = 'mpirun␣-np␣4␣{}'

executables = json.loads(argv[1])
nSamples = int(argv[2])

if not os.path.isfile("Ctrl"):
    print("Error:␣no␣control␣file␣found!")
    sys.exit(1)

ctrl = {}
with open("./Ctrl") as ctrlFile:
    for line in ctrlFile:
        key, value = line.partition("=")[::2]
        ctrl[key.strip()] = value.strip()

root = os.getcwd()

for n in [0.2]:
    for client, exe in executables.items():
        os.chdir(root)

        test_id = "{}_{}".format(client, n)

        if not os.path.isdir(test_id):
            os.mkdir(test_id)

        ctrl["stop_time"] = n
        with open("{}/Ctrl".format(test_id), 'w') as f:
            for k, v in ctrl.items():
                f.write("{}␣=␣{}\n".format(k, v))

        e = os.path.join(os.path.abspath(os.getcwd()), exe)
        os.chdir(test_id)
        reportFile = os.path.join(os.path.abspath(os.getcwd()), "report")

        with open(reportFile, 'a') as f:
            f.write(MPICommand.format(e) + "\n")

        for i in range(nSamples):
            print("Running␣test␣id={},␣i={}".format(test_id, i))
            ts = time.time()
            subprocess.call(MPICommand.format(e), shell=True, stdout=subprocess.DEVNULL)
            te = time.time()
            with open(reportFile, 'a') as f:
                f.write("{}\n".format(te - ts))
```

Listing B.6: `benchmarks/lj/benchmark.py` — Benchmark script for the Lennard-Jones case study.

```
client lennard_jones__single_phase_
  add_arg(sigma, <#real(mk)#>, default=1.0_mk, min=0.0_mk, ctrl_name='sigma',
      ↪ help_txt='distance␣of␣potential␣well')
  add_arg(epsilon, <#real(mk)#>, default=1.0_mk, min=0.0_mk, ctrl_name='epsilon',
      ↪ help_txt='potential␣well␣depth')
  add_arg(mass, <#real(mk)#>, default=1.0_mk, min=0.0_mk, ctrl_name='mass',
      ↪ help_txt='mass␣of␣particles')
  add_arg(delta_t, <#real(mk)#>, ctrl_name='delta_t', help_txt='time␣step')

  real(ppm_kind_double) :: cutoff
  real(ppm_kind_double) :: skin
  real(ppm_kind_double) :: E_prc
  integer, dimension(6) :: bcdef = ppm_param_bcdef_periodic
  real(ppm_kind_double) :: Ev_tot
  real(ppm_kind_double) :: Ep_tot
  real(ppm_kind_double) :: E_tot
  real(ppm_kind_double) :: Ev_tot_old
  real(ppm_kind_double) :: Ep_tot_old
  real(ppm_kind_double), dimension(3) :: r_pq
  real(ppm_kind_double), dimension(3) :: dF
  real(ppm_kind_double) :: r_s_pq2
  integer :: st

  ppm_init()


! begin phase all-in-one

  cutoff = sigma * (2.5_mk / 1.1_mk)
  skin = 0.1_mk * cutoff
  E_prc = 4.0_mk * epsilon * (((sigma / (cutoff + skin))**12) - ((sigma / (cutoff +
      ↪  skin))**6))

  topo = create_topology(bcdef, ghost_size=<#cutoff + skin#>)

  ! -- creating particle set --------------------
  parts = create_particles(topo, ghost_size=<#cutoff + skin#>)


  call parts%apply_bc(info)
  global_mapping(parts, topo)

  v = create_property(parts, 3, "velocity", zero=true)
  a = create_property(parts, 3, "acceleration", zero=true)
  F = create_property(parts, 3, "force", zero=true)
  E = create_property(parts, 1, "energy", zero=true)

  ghost_mapping(parts)
  ! -- end: creating particle set ----------------

  nlist = create_neighlist(parts, skin=<#skin#>, cutoff=<#cutoff#>, sym=<#.false
      ↪ .#>)

  call parts%apply_bc(info)



  st = 0

  !print([E=>parts, v=>parts, F=>parts], 100)
  t = timeloop()
    foreach p in particles(parts) with positions(x, writex=true) sca_props(E)
        ↪ vec_props(F, a, v)
      a_p(:) = F_p(:) / mass
      x_p(:) = x_p(:) + v_p(:) * delta_t + 0.5_mk * a_p(:) * (delta_t**2)
      F_p(:) = 0.0_mk
      E_p = 0.0_mk
    end foreach

    call parts%apply_bc(info)
    partial_mapping(parts)
    ghost_mapping(parts)
    comp_neighlist(parts)

    foreach p in particles(parts) with positions(x) sca_props(E) vec_props(F)
      foreach q in neighbors(p, nlist) with positions(x)
        r_pq(:) = x_p(:) - x_q(:)
        r_s_pq2 = (r_pq(1)**2) + (r_pq(2)**2) + (r_pq(3)**2)
        dF(:) = (24.0_mk * epsilon * r_pq(:)) * (2.0_mk * ((sigma**12) / (r_s_pq2
            ↪ **7)) - ((sigma**6) / (r_s_pq2**4)))
        F_p(:) = F_p(:) + dF(:)
        E_p = E_p + 4 * epsilon * ((sigma**12) / (r_s_pq2**6) - (sigma**6) / (
            ↪ r_s_pq2**3)) - E_prc
      end foreach
    end foreach

    foreach p in particles(parts) with positions(x) vec_props(F, a, v)
      v_p(:) = v_p(:) + 0.5_mk * (a_p(:) + F_p(:) / mass) * delta_t
    end foreach
    t = t + delta_t

    ghost_mapping(parts)

    Ev_tot_old = Ev_tot
    Ep_tot_old = Ep_tot
    E_tot = 0.0_mk
    Ev_tot = 0.0_mk
    Ep_tot = 0.0_mk
    foreach p in particles(parts) with positions(x) sca_props(E) vec_props(v)
      Ev_tot = Ev_tot + 0.5_mk * mass * ((v_p(0)**2) + (v_p(1)**2) + (v_p(2)**2))
      Ep_tot = Ep_tot + E_p
    end foreach
    Ep_tot = Ep_tot * 0.5_mk
    E_tot = Ev_tot + Ep_tot

    !write(*, '(I7, 3E17.8)'), st, E_tot, Ev_tot, Ep_tot
    !print([E=>parts, v=>parts, F=>parts], 100)

    st = st + 1
  end timeloop
  !print([E=>parts, v=>parts, F=>parts], 100)
! end phase all-in-one

  ppm_finalize()
end client
```

Listing B.7: `benchmarks/lj/orig/lj.ppm` — Original PPM client code for the Lennard-Jones case study as used in the benchmark.

```
client lennard_jones__single_phase_
  add_arg(sigma, <#real(mk)#>, default=1.0_mk, min=0.0_mk, ctrl_name='sigma',
      ↪ help_txt='distance␣of␣potential␣well')
  add_arg(epsilon, <#real(mk)#>, default=1.0_mk, min=0.0_mk, ctrl_name='epsilon',
      ↪ help_txt='potential␣well␣depth')
  add_arg(mass, <#real(mk)#>, default=1.0_mk, min=0.0_mk, ctrl_name='mass',
      ↪ help_txt='mass␣of␣particles')
  add_arg(delta_t, <#real(mk)#>, ctrl_name='delta_t', help_txt='time␣step')

  real(ppm_kind_double) :: cutoff
  real(ppm_kind_double) :: skin
  real(ppm_kind_double) :: E_prc
  integer, dimension(6) :: bcdef = ppm_param_bcdef_periodic
  real(ppm_kind_double) :: Ev_tot
  real(ppm_kind_double) :: Ep_tot
  real(ppm_kind_double) :: E_tot
  real(ppm_kind_double) :: Ev_tot_old
  real(ppm_kind_double) :: Ep_tot_old
  real(ppm_kind_double), dimension(3) :: r_pq
  real(ppm_kind_double), dimension(3) :: dF
  real(ppm_kind_double) :: r_s_pq2
  integer :: st

  ppm_init()

! begin phase all-in-one

  cutoff = sigma * (2.5_mk / 1.1_mk)
  skin = 0.1_mk * cutoff
  E_prc = ((((4.0_mk * epsilon) * (sqrt(((sigma / (cutoff + skin))**12))))) +
      ↪ ((4.0_mk * epsilon) * (sqrt(((sigma / (cutoff + skin)) * ((sigma / (cutoff
      ↪  + skin)) * ((sigma / (cutoff + skin)) * ((sigma / (cutoff + skin)) * ((
      ↪ sigma / (cutoff + skin)) * (sigma / (cutoff + skin)))))))))))) * ((((((sqrt
      ↪ (((sigma / (cutoff + skin))**12))))**1 / 3))**3)) - (sqrt(((sigma / (
      ↪ cutoff + skin)) * ((sigma / (cutoff + skin)) * ((sigma / (cutoff + skin))
      ↪ * ((sigma / (cutoff + skin)) * ((sigma / (cutoff + skin)) * (sigma / (
      ↪ cutoff + skin))))))))))

  topo = create_topology(bcdef, ghost_size=<#cutoff + skin#>)

  ! -- creating particle set --------------------
  parts = create_particles(topo, ghost_size=<#cutoff + skin#>)

  call parts%apply_bc(info)
  global_mapping(parts, topo)

  v = create_property(parts, 3, "velocity", zero=true)
  a = create_property(parts, 3, "acceleration", zero=true)
  F = create_property(parts, 3, "force", zero=true)
  E = create_property(parts, 1, "energy", zero=true)

  ghost_mapping(parts)
  ! -- end: creating particle set ----------------

  nlist = create_neighlist(parts, skin=<#skin#>, cutoff=<#cutoff#>, sym=<#.false
      ↪ .#>)

  call parts%apply_bc(info)

  st = 0

  !print([E=>parts, v=>parts, F=>parts], 100)
  t = timeloop()
    foreach p in particles(parts) with positions(x, writex=true) sca_props(E)
        ↪ vec_props(F, a, v)
      a_p(:) = F_p(:) / mass
      x_p(:) = (1 * (((delta_t * a_p(:)) * (delta_t * 0.5_mk)) + ((v_p(:) * delta_t
          ↪ ) + x_p(:))))
      F_p(:) = 0.0_mk
      E_p = 0.0_mk
    end foreach

    call parts%apply_bc(info)
    partial_mapping(parts)
    ghost_mapping(parts)
    comp_neighlist(parts)

    foreach p in particles(parts) with positions(x) sca_props(E) vec_props(F)
      foreach q in neighbors(p, nlist) with positions(x)
        r_pq(:) = x_p(:) - x_q(:)
        r_s_pq2 = (r_pq(1)**2) + (r_pq(2)**2) + (r_pq(3)**2)
        if (((((sigma**12)) / (r_s_pq2 * (r_s_pq2 * (r_s_pq2 * (r_s_pq2 * (r_s_pq2
            ↪ * (r_s_pq2 * r_s_pq2))))))) < 2.487388931824139d-282)) then
          dF(:) = ((-(24.0_mk * epsilon)) * (r_pq(:) * (((sqrt(((sigma * (sigma * (
              ↪ sigma * (sigma * (sigma * sigma))))) / (r_s_pq2 * (r_s_pq2 * (
              ↪ r_s_pq2 * r_s_pq2))))))**2))))
        else
          dF(:) = (((24.0_mk * epsilon) * r_pq(:)) * ((2.0_mk * (((sigma**12)) / (
              ↪ r_s_pq2 * (r_s_pq2 * (r_s_pq2 * (r_s_pq2 * (r_s_pq2 * (r_s_pq2 *
              ↪ r_s_pq2)))))))) - 0))
        end if
        F_p(:) = F_p(:) + dF(:)
        if ((r_s_pq2 < -2.5113679430174213d-57)) then
          E_p = ((E_p - E_prc) + (-((4 / epsilon) * ((sigma * (sigma * (sigma * (
              ↪ sigma * (sigma * sigma))))) / ((r_s_pq2**3))))))
        else
          if ((r_s_pq2 < 2.2370121346542206d+46)) then
            E_p = ((E_p - E_prc) + ((((-epsilon) * (sigma * (sigma * (sigma * (
                ↪ sigma * (sigma * sigma)))))) / (1 / ((r_s_pq2**3)))) * 4))
          else
            E_p = ((E_p - E_prc) + (-((4 / epsilon) * ((sigma * (sigma * (sigma * (
                ↪ sigma * (sigma * sigma))))) / ((r_s_pq2**3))))))
          end if
        end if
      end foreach
    end foreach

    foreach p in particles(parts) with positions(x) vec_props(F, a, v)
      v_p(:) = v_p(:) + 0.5_mk * (a_p(:) + F_p(:) / mass) * delta_t
    end foreach
    t = t + delta_t

    ghost_mapping(parts)

    Ev_tot_old = Ev_tot
    Ep_tot_old = Ep_tot
    E_tot = 0.0_mk
    Ev_tot = 0.0_mk
    Ep_tot = 0.0_mk
    foreach p in particles(parts) with positions(x) sca_props(E) vec_props(v)
      Ev_tot = Ev_tot + 0.5_mk * mass * ((v_p(0)**2) + (v_p(1)**2) + (v_p(2)**2))
      Ep_tot = Ep_tot + E_p
    end foreach
    Ep_tot = Ep_tot * 0.5_mk
    E_tot = Ev_tot + Ep_tot

    !write(*, '(I7, 3E17.8)'), st, E_tot, Ev_tot, Ep_tot
    !print([E=>parts, v=>parts, F=>parts], 100)

    st = st + 1
  end timeloop
  !print([E=>parts, v=>parts, F=>parts], 100)
! end phase all-in-one

  ppm_finalize()
end client
```

Listing B.8: benchmarks/lj/herbie/lj.ppm — Generated PPM client code for the Lennard-Jones case study as used in the benchmark.

List of Figures

List of Listings

List of Tables

List of Abbreviations

APEG	Abstract Program Equivalence Graph
AST	Abstract Syntax Tree
DSL	Domain-specific Language
EMF	Eclipse Modeling Framework
EPEG	Equivalence Program Expression Graphs
GPL	General Purpose Language
GS	Gray-Scott
HPC	High-Performance Computing
IDE	Integrated Development Environment
IR	Intermediate Representation
LJ	Lennard-Jones
LOP	Language-Oriented Programming
MPS	Meta Programming System
ODE	Ordinary Differential Equation
PA	Particle Access
PDE	Partial Differential Equation
PLA	Particle-List Access
PPM	Parallel Particle Mesh library
PPML	PPM Language
XML	Extensible Markup Language

Bibliography

[ADS10] O. Awile, Ö. Demirel, and I. F. Sbalzarini. "Toward an object-oriented core of the PPM library". In: *Proceedings of ICNAAM: International Conference of Numerical Analysis and Applied Mathematics*. 2010, pp. 1313–1316.

[AS16] Y. Afshar and I. F. Sbalzarini. "A Parallel Distributed-Memory Particle Method Enables Acquisition-Rate Segmentation of Large Fluorescence Microscopy Images". In: *PLoS ONE* 11(4), (2016).

[Aus06] M. A. Austin. "Matrix and finite element stack machines for structural engineering computations with units". In: *Advances in Engineering Software* 37(8), (2006), pp. 544–559.

[Awi+13] O. Awile et al. "A domain-specific programming language for particle simulations on distributed-memory parallel computers". In: *Proceedings of Particles: 3rd International Conference on Particle-based Methods. Fundamentals and Applications*. 2013, pp. 436–447.

[Aya15] U. Ayachit. *The ParaView Guide: A Parallel Visualization Application*. Kitware, Inc., 2015.

[BAS13] F. Büyükkeçeci, O. Awile, and I. F. Sbalzarini. "A portable OpenCL implementation of generic particle-mesh and mesh-particle interpolation in 2D and 3D". In: *Parallel Computing* 39(2), (2013), pp. 94–111.

[Bet15] L. Bettini. "Implementing type systems for the IDE with Xsemantics". In: *Journal of Logical and Algebraic Methods in Programming* 1, (2015), pp. 1–26.

[Bür+11] C. Bürger et al. "Reference attribute grammars for metamodel semantics". In: *Software Language Engineering*, (2011), pp. 22–41.

[CFH06] P. Cook, C. Fidge, and D. Hemer. "Well-Measuring Programs". In: *17th Australian Software Engineering Conference 2006 (ASWEC'06)*. 2006, pp. 253–261.

[CG88] R. F. Cmelik and N. H. Gehani. "Dimensional Analysis with C++". In: *IEEE Software* 5(May), (1988), pp. 21–27.

[Che60] T. E. Cheatham. "Handling fractions and n-tuples in algebraic languages". In: *Communications of the ACM* 3(7), (1960), pp. 391–391.

[Clé+86] D. Clément et al. “A Simple Applicative Language: Mini-ML”. In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, (1986), pp. 13–27.

[CPS12] J. Cardinale, G. Paul, and I. F. Sbalzarini. “Discrete Region Competition for Unknown Numbers of Connected Regions”. In: *IEEE Trans. Image Processing* 21(8), (2012), pp. 3531–3545.

[DD94] C. J. Date and H. Darwen. *A guide to the SQL standard: a user’s guide to the standard relational language SQL*. Addison-Wesley, 1994.

[DHK84] V. Donzeau-Gouge, G. Huet, and G. Kahn. “Programming Environments based on Structured Editors: The MENTOR Experience”. In: *Interactive Programming Environments*. 1984, pp. 128–140.

[DK98] A. van Deursen and P. Klint. “Little languages: little maintenance?” In: *Journal of Software Maintenance* 10(2), (1998), pp. 75–92.

[DKV00] A. van Deursen, P. Klint, and J. Visser. “Domain-specific languages: an annotated bibliography”. In: *ACM Sigplan Notices* 35(June), (2000), pp. 26–36.

[Dmi04] S. Dmitriev. “Language Oriented Programming: The Next Programming Paradigm”. In: *JetBrains onBoard* (November), (2004).

[EB10] M. Eysholdt and H. Behrens. “Xtext: implement your language faster than the quick and dirty way”. In: *Proceedings of OOPSLA ’10: Conference on Object-Oriented Programming Systems, Languages, and Applications*. 2010, pp. 307–309.

[EH07] T. Ekman and G. Hedin. “The JastAdd Extensible Java Compiler”. In: *ACM SIGPLAN Notices* 42(10), (2007).

[Erd+13] S. Erdweg et al. “The State of the Art in Language Workbenches”. In: *Software Language Engineering SE - 11*. 2013, pp. 197–217.

[Fow05] M. Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. URL: `http://martinfowler.com/articles/languageWorkbench.html` (visited on 07/27/2016).

[Fow10] M. Fowler. *Domain-specific Languages*. Addison-Wesley Professional, 2010.

[FS01] D. Frenkel and B. Smit. *Understanding molecular simulation : from algorithms to applications*. Elsevier, 2001.

[Geh77] N. H. Gehani. “Units of Measure as a Data Attribute”. In: *Comput. Lang.* 2(3), (1977), pp. 93–111.

[GLT99] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, 1999.

[Gre+15] S. Grewe et al. “Type Systems for the Masses: Deriving Soundness Proofs and Efficient Checkers”. In: (2015), pp. 137–150.

[GS84] P. Gray and S. K. Scott. “Autocatalytic reactions in the isothermal, continuous stirred tank reactor. Oscillations and instabilities in the system A + 2B -> 3B; B -> C”. In: *Chemical Engineering Science* 39(6), (1984), pp. 1087–1097.

[Hei+11] F. Heidenreich et al. “Model-Based Language Engineering with EMFText”. In: *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011. Revised Papers*. 2011, pp. 322–345.

[Her16] Herbie - Documentation. *Herbie Input Format*. 2016. URL: `http://herbie.uwplse.org/doc/input.html` (visited on 04/23/2016).

[HM95] I. J. Hayes and B. P. Mahony. “Using units of measurement in formal specifications”. In: *Formal Aspects of Computing* 7(3), (1995), pp. 329–347.

[Hoc70] R. W. Hockney. “The potential calculation and some applications”. In: *Methods in Computational Physics* 9, (1970), pp. 135–211.

[Hou83] R. T. House. “A Proposal for an Extended Form of Type Checking of Expressions”. In: *The Computer Journal* 26(4), (1983), pp. 366–374.

[Hsu+08] S. H. Hsu et al. “A domain-specific compiler theory based framework for automated reaction network generation”. In: *Computers and Chemical Engineering* 32(10), (2008), pp. 2455–2470.

[Hud96] P. Hudak. “Building Domain-specific Embedded Languages”. In: *ACM Comput. Surv.* 28(December), (1996).

[IEE08] IEEE Standards Committee. *754-2008 IEEE standard for floating-point arithmetic*. 2008.

[IM12] A. Ioualalen and M. Martel. “A new abstract domain for the representation of mathematically equivalent expressions”. In: *Static Analysis*, (2012).

[Kar+15] S. Karol et al. *Towards a Next-Generation Parallel Particle-Mesh Language*. 2015.

[Ken94] A. Kennedy. “Dimension Types”. In: *Esop* 788, (1994), pp. 348–362.

[Ken97] A. J. Kennedy. “Relational parametricity and units of measure”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’97* 1(January), (1997), pp. 442–455.

[KK98] G. Karypis and V. Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM Journal on Scientific Computing* 20(1), (1998), pp. 359–392.

[KL78] M. Karr and D. B. Loveman. “Incorporation of units into programming languages”. In: *Communications of the ACM* 21(5), (1978), pp. 385–391.

[Kos+12] G. Kossakowski et al. “JavaScript as an Embedded DSL”. In: *ECOOP 2012 – Object-Oriented Programming: 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. 2012, pp. 409–434.

[KV10] L. C. Kats and E. Visser. “The Spoofax Language Workbench”. In: *ACM SIGPLAN Notices* 45(10), (2010).

[LMB92] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O’Reilly, 1992.

[LR11a] D. H. Lorenz and B. Rosenan. “Cedalion: a language for language oriented programming”. In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and*

Applications, OOPSLA 2011, part of SPLASH 2011. (Oct. 22, 2011). 2011, pp. 733–752.

[LR11b] D. H. Lorenz and B. Rosenan. “Code Reuse with Language Oriented Programming”. In: *12th International Conference on Software Reuse, ICSR 2011*. 926. 2011, pp. 167–182.

[Lup+15] F. Luporini et al. “Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly”. In: *ACM Transactions on Architecture and Code Optimization* 11(4), (2015).

[LW91] M. S. Lam and M. E. Wolf. “A Data Locality Optimizing Algorithm”. In: *Proceedings of the ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation*. 4. 1991.

[Mar06] M. Martel. “Semantics of roundoff error propagation in finite precision calculations”. In: *Higher-Order and Symbolic Computation* 19(1), (2006), pp. 7–30.

[Mar09] M. Martel. “Program Transformation for Numerical Precision”. In: *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. March. 2009, pp. 101–110.

[MHS05] M. Mernik, J. Heering, and A. M. Sloane. “When and how to develop domain-specific languages”. In: *ACM Computing Surveys* 37(4), (2005), pp. 316–344.

[Mon92] J. J. Monaghan. “Smoothed Particle Hydrodynamics”. In: *Annual Review of Astronomy and Astrophysics* 30, (1992), pp. 543–574.

[MPS14] MPS - 3.2 - Documentation. *Cookbook - Type System*. 2014. URL: `https://confluence.jetbrains.com/display/MPSD32/Cookbook+-+Type+System` (visited on 03/22/2016).

[MPS15a] MPS - 3.2 - Documentation. *Cookbook - Generator*. 2015. URL: `https://confluence.jetbrains.com/display/MPSD32/Generator+cookbook` (visited on 06/22/2016).

[MPS15b] MPS - 3.2 - Documentation. *TextGen*. 2015. URL: `https://confluence.jetbrains.com/display/MPSD32/TextGen` (visited on 07/14/2016).

[MPS15c] MPS - 3.2 - Documentation. *Typesystem*. 2015. URL: `https://confluence.jetbrains.com/display/MPSD32/Typesystem`.

[MPS15d] MPS - 3.2 - Documentation. *User’s Guide*. 2015. URL: `https://confluence.jetbrains.com/display/MPSD32/MPS+User's+Guide` (visited on 04/24/2016).

[Nar93] B. A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT Press, 1993.

[NCM03] N. Nystrom, M. Clarkson, and a.C. Myers. “Polyglot: An extensible compiler framework for Java”. In: *Proceedings of the 12th International Conference on Compiler Construction* (April), (2003), pp. 138–152.

[Öqv12] J. Öqvist. “Implementation of Java 7 Features in an Extensible Compiler”. Master’s Thesis. Lund University, 2012.

[Pan+15] P. Panchekha et al. “Automatically Improving Accuracy for Floating Point Expressions”. In: *PLDI’15*. 6. 2015, pp. 1–11.

[Par13] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[Plo04] G. D. Plotkin. “The origins of structural operational semantics”. In: *Journal of Logic and Algebraic Programming* 60-61(SUPPL.), (2004), pp. 3–15.

[Plo81] G. D. Plotkin. “A Structural Approach to Operational Semantics”. In: *Techreport* (DAIMI FN-19), (1981).

[Rat+12] D. Ratiu et al. “Implementing modular domain specific languages and analyses”. In: *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation - MoDeVVa ’12*. 2012, pp. 35–40.

[RO12] T. Rompf and M. Odersky. “Lightweight Modular Staging”. In: *Communications of the ACM* 55(6), (2012), p. 121.

[Ros10] B. Rosenan. “Designing language-oriented programming languages”. In: *Proceedings of OOPSLA ’10: Conference on Object-Oriented Programming Systems, Languages, and Applications*. 2010, pp. 207–208.

[Sam69] J. E. Sammet. *Programming languages: history and fundamentals*. Prentice-Hall, 1969.

[Sba+06] I. F. Sbalzarini et al. “PPM - A highly efficient parallel particle-mesh library for the simulation of continuum systems”. In: *Journal of Computational Physics* 215(2), (2006), pp. 566–588.

[Sba09] I. F. Sbalzarini. “Abstractions and Middleware for Petascale Computing and Beyond”. In: *International Journal of Distributed Systems and Technologies* 1(2), (2009), pp. 40–56.

[SCC06] C. Simonyi, M. Christerson, and S. Clifford. “Intentional software”. In: *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006.* (Oct. 22, 2006). 2006, pp. 451–464.

[SH11] E. Söderberg and G. Hedin. “Building semantic editors using JastAdd: tool demonstration”. In: *Proceedings of LDTA ’11: Language Descriptions, Tools and Applications*. (Mar. 26, 2011). 2011.

[Tat+11] R. Tate et al. “Equality Saturation: A New Approach To Optimization”. In: *Logical Methods in Computer Science* 7(1), (2011), pp. 1–37.

[Umr94] Z. D. Umrigar. “Fully static dimensional analysis with C++”. In: *ACM SIGPLAN Notices* 29(9), (1994), pp. 135–139.

[Ver67] L. Verlet. “Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules”. In: *Phys. Rev.* 159(1), (1967), pp. 98–103.

[Voe+12] M. Voelter et al. “mbeddr: an extensible C-based programming language and IDE for embedded systems”. In: *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity - SPLASH ’12*. 2012.

[Voe+13] M. Voelter et al. “mbeddr: Instantiating a language workbench in the embedded software domain”. In: *Automated Software Engineering* 20(3), (2013), pp. 339–390.

[Voe11] M. Voelter. “Language and IDE Modularization and Composition with MPS”. In: *Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*. 2011, pp. 383–430.

[Voe14] M. Voelter. “Generic Tools, Specific Languages”. PhD thesis. TU Delft, Delft University of Technology, 2014.

[War94] M. P. Ward. “Language Oriented Programming”. In: *Software Concepts and Tools* 15(4), (1994), pp. 147–161.

[Wex81] R. L. Wexelblat. *History of Programming Languages*. 1981.

[WO91] M. Wand and P. O’Keefe. “Automatic Dimensional Inference”. In: *Computational Logic - Essays in Honor of Alan Robinson*, (1991), pp. 479–483.

Disk Content

The enclosed disk contains the source code implementations presented in this document and the document source itself. The content is structured as follows:

File/Directory	Content
`2016_DA_Nett.pdf`	The PDF file of this document.
`readme.txt`	A readme file with general information about the usage of the contained files.
`doc/`	The LaTeX sources of this document, including all figures.
`doc/img/`	The graphics and figures used throghout this document.
`ppme/`	The source code of PPME.
`benchmark/`	The benchmark setups and results.