Pattern Matching for an Object-oriented and Dynamically Typed Programming Language

Felix Geller, Robert Hirschfeld, Gilad Bracha

Technische Berichte Nr. 36

des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam | 36

Felix Geller | Robert Hirschfeld | Gilad Bracha

Pattern Matching for an Object-oriented and Dynamically Typed Programming Language

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Universitätsverlag Potsdam 2010
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 4623 / Fax: 3474
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam**.**

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2010/4303/
URN urn:nbn:de:kobv:517-opus-43035
http://nbn-resolving.org/urn:nbn:de:kobv:517-opus-43035

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-065-6

Abstract

Pattern matching is a well-established concept in the functional programming community. It provides the means for concisely identifying and destructuring values of interest. This enables a clean separation of data structures and respective functionality, as well as dispatching functionality based on more than a single value. Unfortunately, expressive pattern matching facilities are seldomly incorporated in present object-oriented programming languages.

We present a seamless integration of pattern matching facilities in an object-oriented and dynamically typed programming language: Newspeak. We describe language extensions to improve the practicability and integrate our additions with the existing programming environment for Newspeak.

This report is based on the first author's master's thesis.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Programming languages are important tools for software developers. They are used to express solutions as general instructions to a machine. Defining algorithms in machine language can be cumbersome. For this reason high-level programming languages attempt to hide the details of the underlying machine to a certain degree. For this purpose they are often modeled around a pattern of thought for expressing a solution.

The programming language Smalltalk [GR83] follows an object-oriented model where the basic data type is an object and functionality is defined by sending messages between different objects. The language Haskell [Jon03] supports solving problems following a functional approach in a mathematical sense: By defining the result of applying a procedure to a set of arguments. While these two examples display rather distinct approaches, language designers may choose to combine the features of different paradigms. Ultimately, it is up to the software engineer to pick a language that is a best fit for the task at hand.

The features of a language reflect the underlying paradigm. Each language supports a core set of features which facilitate solving problems according to the involved paradigm, such as an object-oriented model. Smalltalk supports a class-based system to describe a family of objects while Haskell provides for higher-order functions to model a mathematical approach. For actively used programming languages, this set of features is subject to constant change.

Change might be driven by the need of the users of a language. However, care should be taken to adhere to the language's original pattern of thought. One example is the request for support of closures for the Java™ programming language [GJSB05]. The language was designed based on an object-oriented model and its community only recently identified the need for functions as values. Issues like a non-local return have raised a lively discussion on whether the addition would fit with the language as a whole.

Additions to a language should improve its expressiveness. New language constructs should be orthogonal to existing features. Rather than "piling feature on top of feature" [SDF+09], language designers should consider cutting redundant components and take care when accepting an extension. It is generally beneficial to implement additions to a languages as a library. It preserves compatibility with the language core and facilitates the adaption by users to provide for specialized versions. For example, the language Smalltalk supports blocks which enable the implementation of control structures without extending the set of keywords of the language or modifying the compiler. It is a tribute to the language design itself and its core features that extensions implemented as libraries can be assimilated into the original language.

The focus of this work is the extension of the programming language Newspeak [Bra09] by pattern matching facilities. We propose a seamless integration of pattern matching facilities in the Newspeak programming language. Given the design guidelines motivated above, we describe the benefits and consider issues that arise from transferring a concept which originated in the functional programming community to an object-oriented language. For example, encapsulation is a concept that is inherent to the object-oriented model and should not be broken by our extensions to the language.

1.1 Contributions

The main goal of our work is the extension of the Newspeak programming language by pattern matching facilities. To that effect, the central contributions of this work are the following:

A Model for Pattern Matching Given the background on pattern matching in functional programming languages, we extract a common basis in order to transfer the concept to an object-oriented programming language. We present a basic yet flexible model which allows for expressing most pattern matching facilities known from existing languages.

Pattern Matching for Newspeak Based on the introduced model, we present a complete implementation for the Newspeak programming language. We provide further extensions based on the described core in order to facilitate using the new facilities, for example, keyword patterns that allow for an intuitive way to inspect and expose an object's content, and language literals for the host language that reduce the overhead for using the presented pattern matching facilities.

Application Developer Tool Support We present extensions to the existing Newspeak programming environment to provide for the employment of our pattern matching facilities in application development. Our work integrates an augmented compiler with the existing environment. This enables application developers to use existing tools such as the environment's system browser or debugger.

1.2 Report Structure

The rest of this work is structured as follows. Background knowledge on pattern matching and our design objectives for extending Newspeak are provided in the following section. The sections 3, 4 and 5 describe the main contributions of this work. Section 3 presents a model for adding pattern matching facilities to an object-oriented language. Section 4 describes the implementation of this model in Newspeak as well as further extensions to the language which aim to increase the usability of the new pattern matching facilities. Section 5 presents the additions to the Newspeak programming environment. Next, we evaluate our work with respect to the outlined objectives as well as related techniques. Related work is presented in Section 7, before we conclude this work by providing a summary and future work.

2 Background

Pattern matching is a well known concept that originates in the functional programming community. This section provides background knowledge on pattern matching facilities in traditional functional programming languages. We use the well-established programming language Haskell to illustrate common usage of pattern matching. Simple examples are used to highlight the benefits of this concept. We describe what existing techniques are generally used when lacking pattern matching facilities in object-oriented programming languages. Furthermore, we discuss the relation of pattern matching and object-orientation and programming languages with a dynamic type system. We conclude this section by presenting the objectives which motivate our design decisions for extending the Newspeak programming language.

2.1 Pattern Matching

Pattern matching facilities were first developed for functional programming languages [Bur69]. Today it is a well established feature and is part of mature languages such as Haskell or members of the ML family [MTH90, Ler98]. The original work on pattern matching in [Bur69] describes syntactic extensions to a functional programming language that facilitate the definition of programs by means of structural induction. Supporting such a technique coincides with the mathematical approach of functional programming languages and the aim to promote equational reasoning. In order to illustrate equational reasoning as well as structural induction, please consider the following two sample functions:

```
-- factorial function
factorial 0 = 1
factorial n = n * factorial (n - 1)

-- length function
len []      = 0
len (hd:tl) = 1 + len tl
```

Listing 1: Examples in Haskell

The first part defines the factorial function for positive integers and the second part a function that returns the length of a list. Both definitions separate the function semantics into two cases: The base case and a more general version (similar to an inductive step). More specifically, the functions are defined recursively over the function's argument.

Both function definitions rely on Haskell's pattern matching facilities. The first part of the factorial function matches the function argument against the number `0` while the second part binds the argument to the name `n` (a name acts as a wildcard of the type `Num`). The base case of the length function matches against the empty list `[]`, while the second part uses the destructuring operator `:` to bind the head and tail of the argument to the names `hd` and `tl` respectively. This allows for applying the `len` function to the tail recursively.

The examples illustrate how Haskell's pattern matching facilities promote equational reasoning by separating the function semantics into distinct parts. However, such separation of function definitions is merely syntactic sugar [Emi07]. The more general case is a language construct which allows for distinguishing multiple cases by matching arguments against patterns. The following listing shows a rewritten version of the length function that employs this construct:

```
-- length function
len lst = case lst of
            []      → 0
            (hd:tl) → 1 + len tl
```

Listing 2: Rewritten Sample Function

The definition in Listing 2 is of equivalent meaning as the version in Listing 1. It combines distinct function bodies into a single definition and employs Haskell's `case of` construct to differentiate the base case from the inductive step. The structure is arguably intuitive: The body of the construct is split into multiple cases which match the given argument `lst` against patterns (either the empty list `[]` or a non-empty list which contains a head and a tail) and specify the respective functionality. Each case resembles a rule that defines an action if a predicate (the pattern) holds.

Our work aims to bring a construct similar to the one presented in Listing 2 to the Newspeak programming language. Splitting method definitions into multiple parts based on arguments at run-time requires a different form of method dispatch from what regular object-oriented programming language such as Newspeak support. Extending the method dispatch procedures of Newspeak is not part of this report.

The following listing involves a slightly more involved example based on [EOW07]. We use it to demonstrate benefits one gains from pattern matching facilities. The first part defines a data structure used to represent a term, which is either a variable, a number or a product of two terms. The second part defines a simplification rule based on the identify element for multiplication of numbers: If the given term is a product whose right side is the number one, then return only the left hand side, otherwise return the entire term.

```haskell
1  -- representing terms
2  data Term a b = Variable a | Number b | Product (Term a b) (Term a b)
3  
4  -- simplification rule
5  simplify term =
6      case term of
7        Product lhs (Number 1) → lhs
8        _                      → term
```

Listing 3: Pattern Matching Example in Haskell Based on [EOW07]

The example illustrates the usage of pattern matching to "inspect and decompose data" [Emi07] in a single expression. The pattern in line 7 describes the desired structure of the given argument and binds the left hand side to the name `lhs`. Regular object-oriented programming languages fail to provide a construct for this purpose. Instead, programmers use regular conditional statements to identify an object and then retrieve values of interest via accessors in a separate step. This is often aggravated in the context of a statically typed programming language such as Java™ (see also, Section 6 and Appendix A), where developers often have to resort to manually "casting" the type of an object in order to access the contained values

Pattern matching facilities often support nested patterns. The example in Listing 3 contains a nested pattern in line 7: We match the right hand side of the product against a pattern for a representation of the number one. In contrast to regular accessors and conditional statements, it can be argued that "deep pattern matching allows concise, read-able deconstruction of complex data structures" [LM03]. More specifically, multiple nested conditional expressions quickly become difficult to read, while nested patterns allow destructuring of nested data containers in a single expression.

Nested patterns also allow for distinguishing cases based on more than a single argument. The implementation of `simplify` illustrates how pattern matching allows for "multiple dispatch". Functionality is dispatched based not only on the type of the `term` argument but also nested values, such as the right hand side being a representation of the number one.

Another benefit gained from pattern matching is a possible separation of data structure and respective functionality. The example in Listing 3 demonstrates a clear division: The first part defines the structure of the data container and the second part defines functionality independently.

However, it is worth noting that the given example fails to provide a clean separation: It breaks the principle of encapsulation. The implementation of `simplify` depends directly on the `Term` datatype. Changing the desired data structure implementation, ultimately entails updating all functions which depend on the type `Term`. This is clearly not desirable for an object-oriented programming language.

The visitor pattern [GHJV95] is a well known technique to the users of object-oriented programming languages. Application developers use it to imitate double dispatch and achieve a separation of a data structure and respective functionality. As highlighted above, pattern matching can be applied for similar purposes. Arguably, pattern matching facilities provide for a more concise and scalable solution [EOW07].

More recent efforts show the desirability of pattern matching facilities for other communities. Several emerging programming languages aim to combine the benefits of pattern matching with those of other paradigms. The Scala programming language for the Java™ Virtual Machine platform includes a general pattern matching construct as part of an effort to "unify object-oriented and functional programming" [OAC+09]. Similarly, the work on the F# programming language [SM] aims to "bridge the gap between the functional and object-oriented worlds" [SNM07] and is targeted at Microsoft's .NET platform.

We have highlighted several benefits of pattern matching and identified related techniques in object-oriented programming languages, which do not support this concept. The goal of this work is the careful extension of an object-oriented programming language by pattern matching facilities in order to gain the highlighted benefits without breaking existing paradigms. We now describe our view of the relation between pattern matching and object-orientation as well as implications when adding it to a dynamically typed language which motivates our work.

2.2 Pattern Matching and Object-Orientation

In pure object-oriented programming languages each value is an object. Hierarchies of objects are used to organize information. For example, consider an abstract syntax tree. Each node of the tree is an object which holds the information relevant to a particular syntax entity. Applications such as interpreters or compilers traverse such data structures and extract the parts of interest.

Traversing a complex data structure can quickly become tedious. Application developers employ techniques such as the visitor pattern to compensate for the shortcomings of the used programming language. More specifically, regular object-oriented programming languages have no inherent construct for identifying an object of interest and destructuring it in a single step.

Pattern matching facilities provide the means to inspect and decompose nested data structures in a single statement. In this regard, we believe that pattern matching is an expressive extension for object-oriented programming languages which organize data in hierarchies of objects. The work in [EOW07] demonstrates how pattern matching facilitates traversing of data structures such as a tree of algebraic expressions.

2.3 Pattern Matching and Dynamic Typing

Pattern matching facilities are often used in statically typed programming languages to provide for “run-time type discrimination without casts” [LM03]. The example in Appendix A illustrates the use of casts in Java™ , which lacks pattern matching facilities. The Haskell version requires no cast as the type inferencer is able to exploit the pattern matching construct to compute the types of the decomposed values.

There is no need to statically declare the type of a value in a dynamic type system. Nevertheless, identifying the type of an object at run-time remains of interest. For example, Smalltalk provides the message `isKindOf:` in order to determine the run-time type of an object. It is used to identify and collect values of interest or dispatch functionality based on the run-time type of a value. A regular and unmodified Squeak Smalltalk image (version 3.10[1]) contains approximately 600 senders of the message `isKindOf:`.

Patterns are used to identify values of interest. This means that objects which match a pattern satisfy the desired properties that are specified by the pattern. These properties might include the desired type of the subject but can also be used for more expressive predicates. For example the simple implementation of the factorial function in Listing 1 matches a function argument against the value `0`.

Dispatching functionality based on types fails the concept of data abstraction. Type discrimination exposes representation and therefore complicates future changes. Pattern matching can offer a cleaner interface for specifying desired characteristics of an object rather by relying on its type. In some sense, pattern matching facilities can be used as a static type system, but are less intrusive and can be more expressive. Hence, we believe that flexible pattern matching facilities are a meaningful extension for a dynamically typed programming language.

2.4 Design Objectives

As briefly outlined in the first section, we believe that language additions should not be redundant. For this purpose, it is important to ensure that language extensions are orthogonal to existing features and increase the overall expressivity. We have highlighted how an object-oriented and dynamically typed programming language may benefit from providing pattern matching facilities. However, transferring a concept from a functional programming language might break with existing language features such as encapsulation.

To assure that existing paradigms are retained, we describe the objectives which motivate our underlying design decisions. We aim for a seamless integration of expressive additions while enabling application developers to conveniently apply our language extensions. Furthermore we attempt to facilitate future change, which motivates the first objective.

Enable Abstraction over Patterns Current statically typed functional programming languages such as Haskell fail to provide the means for abstracting over patterns as values [Tul00]. More specifically, patterns are not first class

[1]Squeak Smalltalk images are available online at http://www.squeak.org/Download/, last accessed: March 14, 2010.

values. This means that it is impossible to extract common functionality from different pattern matching instances. Consider filtering a list of elements based on a pattern: A generic implementation would be parametrized over a pattern. However, this is currently not possibly in Haskell and related programming languages. The following listing is based on an example in [Tul00]. It aims to demonstrate the lack of constructs for abstraction over patterns in Haskell:

```
1  filterVariables aList =
2    case aList of
3      []                  → []
4      (Variable v : tl) → v : filterVariables tl
5      (_ : tl)          → filterVariables tl
6
7  filterNumbers aList =
8    case aList of
9      []                  → []
10     (Number n : tl)   → n : filterNumbers tl
11     (_ : tl)          → filterNumbers tl
```

Listing 4: Need for Abstraction Over Patterns Based on [Tul00]

The functions `filterVariables` and `filterNumbers` are virtually the same but it is impossible to extract the common functionality based on patterns alone. More specifically, the only meaningful difference is the use of the patterns `Variable` and `Number` in lines 4 and 10 respectively. This example illustrates the need for patterns to be first class values of the host language.

Making patterns regular values of the host programming language facilitates future change. Modifying the pattern matching facilities of Haskell generally involves changing the respective language compiler. The current plans for Haskell 2010 demonstrate the shortcomings of such an approach: The language has collected an abundance of pattern matching features over the years, rather than enabling application developers to implement specialized versions [2]. We aim to learn from these debates and provide patterns as first class values.

[2]For example, it is questionable whether features such as "n+k patterns" enable program comprehension and there are plans to eliminate them from future Haskell versions. (cf. http://hackage.haskell.org/trac/haskell-prime/wiki/NoNPlusKPatterns, last accessed: March 14, 2010)

Facilitate Composition Pattern composition means the combination of multiple patterns to a new pattern. For this purpose it is essential to define a clear interface for matching a pattern on an object which can be used programmatically. This interface can be used to create operators which combine patterns in a well-defined way. For example, an alternation operator could be used to produce a pattern which matches an object if any of the given two patterns matches the object.

Enabling the composition of patterns facilitates future changes. It provides for programmatically creating new patterns based on operators on patterns. We aim to provide a basic set of patterns and means to compose them in unanticipated ways by such operators. Furthermore, the set of operators should be extensible by application developers.

Preserve Encapsulation Hiding the implementation of functionality is an essential feature of object-oriented programming. Adhering to this design principle increases a system's robustness by facilitating future changes and reducing undesired dependencies. As described in Section 2.1, traditional functional programming languages often fail to accommodate the principle of encapsulation. We aim to preserve this valuable principle in the context of an object-oriented programming language.

Enable Ease of Use Using pattern matching facilities should be intuitive to users of the programming language. The language extensions should be seamless and allow for concisely expressing patterns and related actions. We aim to reduce the overhead required for employing pattern matching facilities. Furthermore, an intuitive integration with the existing programming environment should enable application developers to conveniently employ the new language extensions.

We highlighted the objectives that motivate the design decisions of our work. The work in [EOW07] aims to evaluate pattern matching in the context of different related techniques, such as the visitor pattern. They compare different solutions for implementing a simplifying rule for an algebraic expression. The presented criteria aim to evaluate the solutions with respect to conciseness and extensibility of subjects and patterns among others. We present an evaluation of our work with respect to these solutions and our objectives in Section 6.

⁂

This section provided the necessary background information on pattern matching facilities in traditional functional programming languages and identified related techniques. We highlighted possible benefits from supporting pattern matching and summarized our design objectives. Next, we present a flexible model for pattern matching that allows the transfer to an object-oriented language.

3 A Model for Pattern Matching

In this section, we introduce a model of pattern matching to highlight the concepts that we use to implement our pattern matching facilities. The presented model is a flexible core that allows the transfer of the pattern matching concept to other paradigms. As our goal is the extension of an object-oriented programming language, we describe the model by means of objects and communication via messages. However, the concepts of patterns, bindings and combinators are generic and can be transferred to other paradigms as well.

We deliberately refrain from using examples of a specific programming language to highlight the genericity of the presented model. For concrete examples that demonstrate the described concepts please refer to Section 4, which contains the description of our realization of this model using the Newspeak programming language.

3.1 Patterns

Patterns are used to specify a set of properties that an object needs to satisfy. They act as predicates on objects, identifying a set of objects that fulfill a desired set of criteria. Figure 1 aims to illustrate the use of a pattern to identify the set of objects which represent numbers (loosely based on the example introduced in Listing 3).

Matching a pattern p on a subject s refers to the process of sending a message to p and passing an argument s. The result of this message send indicates whether s meets the requirements as specified by the pattern p. In the following we use the shorthand $p \xleftarrow{?} s$ to refer to this process. It can be read as follows: "Does the pattern p match the object s?". We use objects and messages to describe our model: The arrow symbolizes a message send to the pattern with the subject s as an argument. The return value of this message send is represented by the ? character above the arrow.

In contrast to a simple boolean predicate, a pattern match result value that indicates success usually exposes the constituent parts of the subject. This enables the destructuring aspect of pattern matching which was described in the previous section. The exposed values can be matched by nested patterns, therewith providing for deep matching on data structures. We introduce the concept of binding objects, for the purpose of representing the result of a pattern match and exposing relevant values.

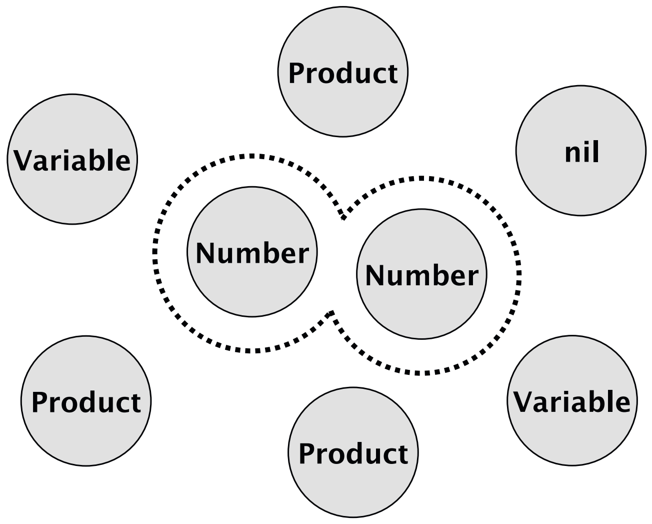


Figure 1: Patterns Identify Objects

3.2 Bindings

Binding objects represent the result of matching a pattern on an object. As the result of the message send $p \xleftarrow{?} s$, they may indicate failure or success depending on whether s meets the requirements specified in p. This completes our requirements for specifying the interface of patterns: $p \xleftarrow{b} s$. The result of $p \xleftarrow{?} s$ is the binding b. More specifically, sending a messages to the pattern p with the argument s evaluates to the binding object b.

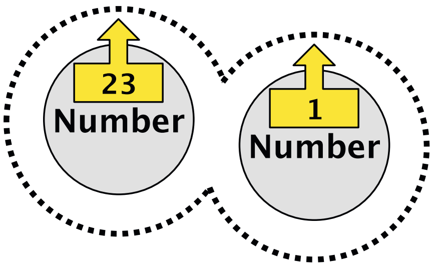


Figure 2: Bindings Expose Nested Data Structures

Binding objects may be used to relay data that is relevant in case of a successful match. Figure 2 aims to illustrate the use of bindings. They are a flag that indicates whether a pattern matches an object and act like a filter that makes certain parts of an object visible.

3.3 Pattern Combinators

Pattern combinators are the means to compose different pattern objects to produce a new pattern. The composition of patterns is based on the common interface of pattern objects $p \xleftarrow{b} s$. Patterns that are the result of invoking pattern combinators must adhere to the pattern interface. Figure 3 aims to illustrate the use of a combinator that acts like a union operator on sets: The composed pattern identifies objects representing variables as well as numbers.

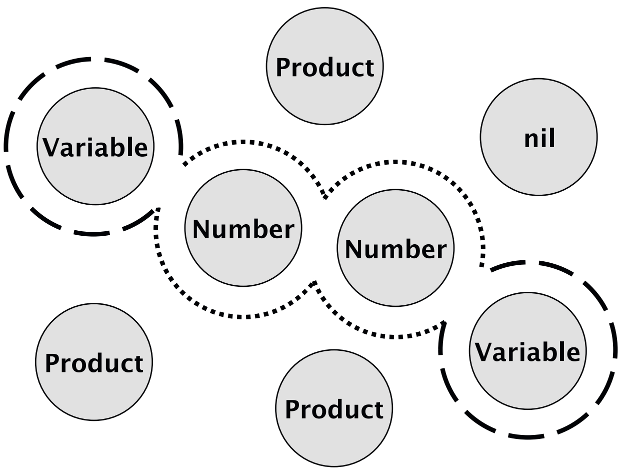


Figure 3: Combinators Enable Composition of Patterns

While we refer to objects and messages, our model for the concepts of pattern matching is not aimed at a particular language. We attempt to point out peculiarities with respect to order of evaluation in the context of combinators. Most pure object-oriented language are imperative and the order of evaluation is usually well-defined. However, we argue it to be possible to implement our model in a purely functional language, where functionality may be evaluated in parallel.

Given the description of pattern combinators and the interface of patterns, we procced to describe a small set of combinators. This list of combinators should not be considered as comprehensive list but instead as a basic set aiming for flexibility.

3.3.1 Application Combinator

The application combinator is used to associate functionality with the success of matching a pattern on an object. It corresponds to the $\rightarrow$ used in previous listings displaying examples of pattern matching in Haskell, for example, Listing 3.

The application combinator $\Rightarrow$ is used to link the evaluation of statements to the success of matching a pattern. Invoking the application combinator on a pattern p_1 with an argument c evaluates to a new pattern: $p_2 = p_1 \Rightarrow c$. The argument c is a closure that expects a single argument. The result b_2 of $p_2 \xleftarrow{b_2} s$ indicates failure if the result b_1 of $p_1 \xleftarrow{b_1} s$ indicates failure. In case b_1 indicates success, then binding b_2 represents the successful evaluation of c given the argument b_1. For instance, b_2 might hold the resulting value of evaluating the closure c given the argument b_1.

3.3.2 Disjunction Combinator

The disjunction combinator is used to link two patterns together such that the resulting pattern matches an object if at least one of the original patterns match the given object. Given that patterns identify sets of patterns, the disjunction operator is comparable to the union operator for sets (see also, Figure 3).

Invoking the disjunction combinator $\cup$ on two patterns results in a new pattern. Consider a pattern $p_3 = p_1 \cup p_2$ and $p_1 \xleftarrow{b_1} s$, as well as $p_2 \xleftarrow{b_2} s$. The binding object b_3 of $p_3 \xleftarrow{b_3} s$ indicates success if any one of b_1 and b_2 indicates success.

There is no explicit order in which the original patterns must be evaluated. However, the disjunction combinator can be used in connection with the application combinator to construct a sequence of patterns and associated functionality, analogously to a set of clauses for Haskell's `case of` construct: $(p_1 \Rightarrow c_1) \cup (p_2 \Rightarrow c_2)$. Concurrent evaluation might lead to confusion for overlapping patterns, while evaluating p_1 before p_2 might arguably be more intuitive.

3.3.3 Conjunction Combinator

The conjunction combinator is used to link two patterns together such that the resulting pattern matches an object if both of the original patterns match the given object. Analogously to the disjunction combinator, the conjunction combinator is comparable to the intersection operator for sets.

Invoking the conjunction combinator $\cap$ on two patterns results in a new pattern. Consider a pattern $p_3 = p_1 \cap p_2$ and $p_1 \xleftarrow{b_1} s$, as well as $p_2 \xleftarrow{b_2} s$. The binding object b_3 of $p_3 \xleftarrow{b_3} s$ indicates success if b_1 and b_2 indicate success.

Similar to the description of the disjunction combinator, there is no explicit restriction on the evaluation order in which the patterns are used. The newly created pattern may short-circuit the evaluation or match both patterns in parallel and synchronize the results. Furthermore, it is left to the implementor to define the contents of b_3. It might bind a boolean value indicating success, or choose to expose other values based on the bindings b_1 and b_2.

3.3.4 Sequencing Combinator

The sequencing combinator is used to link two patterns such that one pattern is parametrized on the result of matching the other pattern on an object. The goal of the sequencing combinator is to provide for chaining multiple patterns together based on the values exposed by a binding. This is different from chaining patterns with the conjunction combinator, which aims to construct a pattern which matches all given patterns on a single subject. The sequencing combinator matches a pattern on an object and passes the resulting binding object to the next pattern (rather than the original subject).

Invoking the sequencing combinator $\gg$ on two patterns yields a new pattern. Consider a pattern $p_3 = p_1 \gg p_2$ and $p_1 \xleftarrow{b_1} s$. The binding object b_3 of $p_3 \xleftarrow{b_3} s$ indicates success if b_2 of $p_2 \xleftarrow{b_2} b_1$ indicates success.

3.3.5 Negation Combinator

The negation combinator is used to invert the result of matching a single pattern on objects. Following the concept that patterns identify sets of objects, the negation combinator corresponds to taking the absolute complement of a set.

Invoking the negation combinator $\nleftarrow$ on a pattern p results in a new pattern $p^C = p \nleftarrow$. The binding object b_c of $p^C \xleftarrow{b_c} s$ is the inverse of the binding b of

$p \xleftarrow{b} s$: The binding b_c indicates success if b indicates failure and vice versa.

⁂

In this section we introduced the concepts of patterns, bindings and combinators. They are the main constituents of the model of pattern matching that we use to extend the object-oriented programming language Newspeak. In the following, we describe our implementation of this model, as well as further extensions to facilitate the matching on regular objects and reduce the overhead of employing our pattern matching facilities.

4 Pattern Matching for Newspeak

We will now introduce our pattern matching facilities for the Newspeak programming language. We base our work on the realization of the model presented in Section 3. We use this flexible core to implement features known from existing pattern matching facilities. Furthermore, we describe our additions that aim to facilitate pattern matching for objects in Newspeak: Keyword patterns. In order to reduce the overhead needed for employing our pattern matching facilities, we introduce extensions to the Newspeak programming language itself: Literals for patterns. We start our presentation by introducing the implementation of literals in order to leverage them for more concise examples in the following parts.

4.1 Literals for Patterns

Before we describe our implementation of the generic model based on patterns, bindings and combinators, we provide an overview of our language extensions. We add support for pattern literals in order to facilitate the usage of our pattern matching facilities by application developers. Augmenting Newspeak by literals mainly involves the following steps: Extending the grammar and respective parser for the Newspeak language and providing for the compilation of the new features.

4.1.1 Extending the Grammar

The grammar for the Newspeak programming language is defined in terms of an executable grammar [Bra07]. Grammar production rules are defined by means of slots of a class and functionality is associated by specializing the respective slot accessor. This provides for expressing a grammar as a program and implementing a parser by subclassing an existing grammar and overriding the respective production rules. Please consider the work in [Bra07] for more information on executable grammars.

The current version of the grammar for the Newspeak language is defined in the `Newspeak2Grammar` module. We extend the grammar by modifying the header of contained `NS2Grammar` class to contain production rules for the new language features. The following listing presents our additions to the existing grammar:

The first two lines are the entry point for our modifications, we add

```
 1  "adding literals for patterns"
 2  literal =
 3    pattern | number | symbolConstant | characterConstant | string | tuple.
 4
 5  pattern =  (tokenFromChar: $<), patternLiteral, (char: $>).
 6  patternLiteral = wildcardPattern | literalPattern | keywordPattern.
 7
 8  wildcardPattern = tokenFromChar: $_.
 9
10  "values which are comparable by means of #="
11  literalPattern = tokenFor:
12    number | symbolConstant | characterConstant | string | tuple.
13
14  "keyword patterns"
15  keywordPattern = kwPatternPair plus.
16  kwPatternPair = keyword, kwPatternValue opt.
17  kwPatternValue =
18    wildcardPattern | literalPattern | variablePattern | nestedPatternLiteral.
19  variablePattern = tokenFor: ( (char: $?), id ).
20  nestedPatternLiteral = tokenFor: ( pattern ).
```

Listing 5: Newspeak Grammar Extension

the `pattern` alternative to the list of literals. Line 5 defines the syntax for pattern literals: They are enclosed by a less-than and greater-than sign (the `tokenFromChar:` and `tokenFor:` methods produce rules which ignore leading white-space). We add support for three different patterns:

1. A wildcard pattern `<_>`,
2. literal value patterns (e.g., `<2>` or `<'name'>`),
3. keyword patterns (e.g., `<name:'Hans' age:23>`).

The wildcard pattern matches any object. The patterns for literal values match objects which are equal to the specified value by means of `=`, this holds for the existing literals such as numbers or strings. Keyword patterns aim to facilitate the matching of arbitrary objects. They consist of patterns which are labeled by keywords, as can be seen in the grammar in lines 15 and 16. We describe them in more detail in Section 4.3.

4.1.2 Extending the Parser

We gave an overview of our extensions to the Newspeak grammar. The executable grammar library provides for a clean separation of the grammar, the parser and the associated abstract syntax tree (AST). In order to enable the compilation of pattern literals, we need to add nodes to the AST to represent pattern literals and modify the parser to instantiate these nodes accordingly.

The `Newspeak2Parsing` module contains the parser corresponding to the grammar defined in `Newspeak2Grammar`. We extend the respective nested `Parser` class (which subclasses `NS2Grammar`) by specializing the accessor methods for the production rules defined above in Listing 5. Consider the following example:

```
"Parser>>"
literalPattern ↑ <LiteralPatternAST> = (
↑ super literalPattern
        wrap: [ :litTok |
                LiteralPatternAST new
                        literal: litTok token;
                        start: litTok start; end: litTok end.
        ].
)
```

Listing 6: Extending The Parser

The method defines the parser for the `literalPattern` production rule. Its single argument is a representation of token for the specified literal value (dropping leading white-space). The purpose of the parser is the production of an AST that represents a pattern for a literal value: `LiteralPatternAST`. We instantiate the respective node and populate it with the literal value, as well as the position of the value in the input stream. The remaining parsers are defined analogously.

4.1.3 Extending the Compiler

We use the compiler framework in the `Newspeak2Compilation` module to implement our language extensions for a variety of reasons: The compiler supports the use of the above described grammar and parser modules and offers a clean front-end. Most importantly though, the new compiler framework reduces the dependence on the Squeak Smalltalk compared to the currently used compiler. It is thus an important step in the evolution of the Newspeak language and we argue that using a deprecated compiler would be futile.

The compiler produces a representation of a method or class that can be installed into the host environment. It translates the AST produced by the parser in multiple passes. Each step of the compilation process is implemented in terms of a visitor which traverses the AST. For example, a visitor is used to produce a mapping for the scope at each node of a given AST.

Our goal is the support of literals for patterns. We enable pattern literals by means of translation: The literals are translated to messages, which will be evaluated like regular message sends at run-time. This strategy is an important difference when compared to other literals such as strings. These values are represented internally by the underlying virtual machine. However, translating literals to message sends the long term goal as it is "closer to the late bound spirit of the language" [Bra09]. More specifically, a message can ultimately be subject to specialization by applications developers.

```
"Rewriter>>"
literalPatternNode: aNode = (
  | patClass literalMessage |

  patClass:: NormalSendAST new
               to: selfNode
               send: (MessageAST new
                        sel: #Pattern;
                        args: {};
                        start: aNode start; end: aNode end).

  literalMessage:: MessageAST new
                     sel: #literal: ;
                     args: { aNode literal };
                     start: aNode start; end: aNode end.

  ↑ (NormalSendAST new
       to: patClass
       send: literalMessage) apply: self.
)
```

Listing 7: Extending The Compiler

We implement the translation of pattern literals by augmenting existing visitors for Newspeak ASTs. For example, the `Rewriter` class is a visitor that is part of the compiler framework in the Newspeak2Compilation module. It is used to convert a given Newspeak AST into an AST which is more suitable for producing a `CompiledMethod`. We add translation rules for the new AST nodes that represent patterns.

Listing 7 illustrates the translation of an AST node produced by the parser presented in Listing 6. More specifically, the literal `<23>` is translated to a message send: `Pattern literal: 23`. The implicit receiver of this message is the root of the object hierarchy of Newspeak: `NewspeakObject` and the result of the message send is an instance, which is a pattern object. We provide for an implementation of the message `Pattern` in `NewspeakObject.` However, it is still a virtual method call, allowing for dynamically changing the semantics of literals.

4.2 A Pattern Matching Model for Newspeak

We describe our realization of the model presented in Section 3 in Newspeak. Our presentation corresponds to the structure of Section 3 and relies on the introduced terminology. We describe the underlying implementation and use examples for illustration purposes.

4.2.1 Patterns

Pattern objects are created by instantiating the class `Pattern` or its subclasses. The interface $\mathtt{p} \xleftarrow{b} s$ is realized by the method `doesMatch:else:` in a pattern object. The method expects two arguments: The first argument is the object s on which the pattern is matched, while the second argument is a closure that is evaluated if the pattern does not match a given object. The resulting binding object `b` is represented by an instance of the class `Binding` or any of its subclasses. Listing 8 is a brief illustration of matching an object 'Peter' on a pattern. Figure 4 highlights the important features of the `Pattern` class. The individual slots and instance methods are described in more detail below. The class methods `wildcard`, `literal:` and `keywords:patterns:` correspond to the literals for patterns that were presented in Section 4.1.

Handling Failure The described interface `doesMatch:else:` makes handling the failure case explicit. We argue this to be especially beneficial in the context

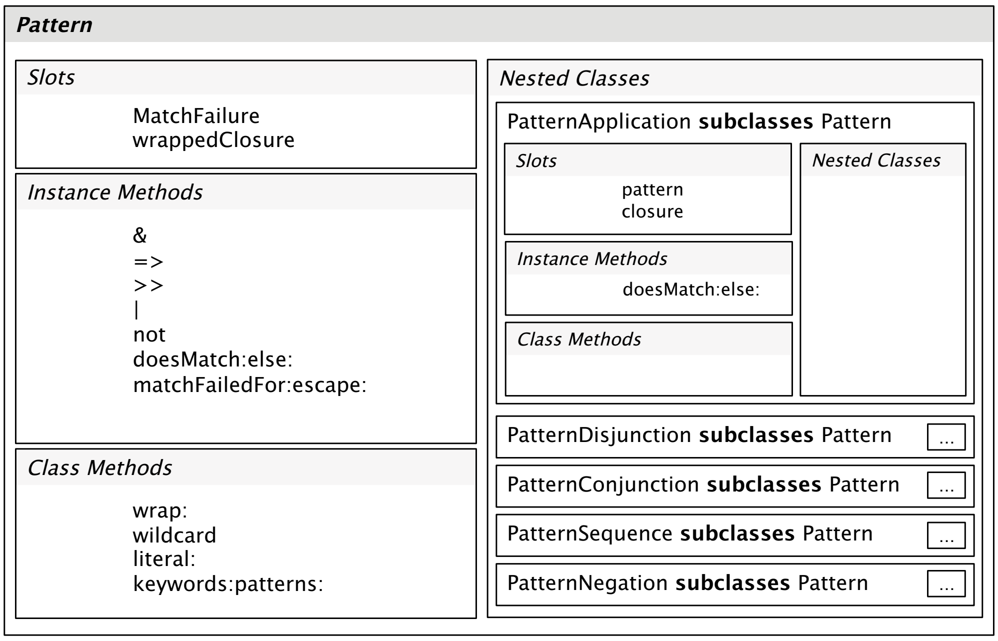


Figure 4: The `Pattern` Class

```
<'Hans'> doesMatch: 'Peter' else: [ Binding new ]
```

Listing 8: Example Illustrating Newspeak Pattern Interface

of non-exhaustive patterns. Furthermore, it is a common idiom to finish a set of clauses with a wildcard pattern similar to an `else` statement in a conditional (cf. Listing 3), which can also be handled by the closure argument.

The described interface implies that evaluating the failure closure should evaluate to a binding object, as shown in Listing 8. However, passing a closure might entail a non-local return. We leave it to the application developer to decide whether a non-local return is an appropriate solution.

For the sake of convenience, we distinguish two cases: the closure which handles the pattern match failure accepts a single or no argument. If the closure accepts a single argument, the subject is passed in, otherwise the closure is evaluated without an argument. The class `Pattern` provides the method named `matchFailedFor:escape:` which implements this functionality.

4.2.2 Bindings

Binding objects are represented by instances of the class `Binding` or its subclasses. Figure 5 highlights the constituent parts of the `Binding` class: It is a container for a single value and provides accessors for verifying whether the `bindee` slot was populated. Sending the message `isBound` to a binding object indicates success or failure of matching a pattern on an object. In order to expose values which are relevant to the properties specified by the pattern, `Binding` provides a slot accessible by sending the message `boundValue`. Unless this slot is set to a specific value by sending the message `boundValue:` to a binding or populating it upon instantiation by using the factory method named `for:` on `Binding`, sending `isBound` to a binding will indicate failure. For instance, evaluating the statement in Listing 8 will evaluate to the binding object instantiated in the closure argument. The binding is empty and therefore indicates failure.

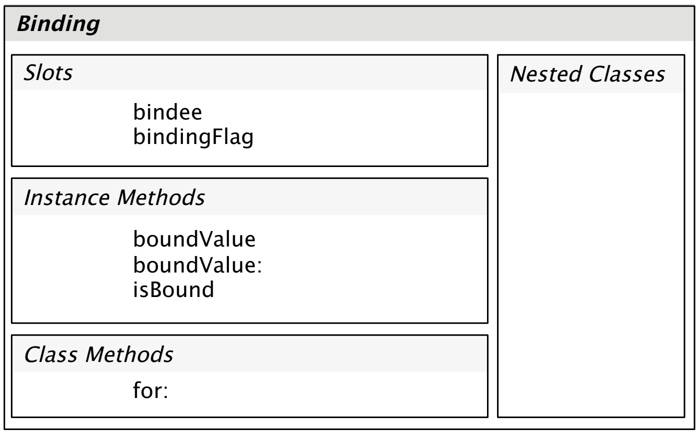


Figure 5: The `Binding` Class

Lines 4 and 5 of Listing 10 demonstrate another idiom to identify the failure of matching a pattern on an object. The class `Pattern` defines a slot named `MatchFailure` which holds an empty binding. Therefore each pattern object holds a specific instance of an empty binding in this slot. Passing this instance to indicate failure allows to compare based on reference rather than sending `isBound` and distinguishing `true` and `false`, while adhering to the common pattern interface.

4.2.3 Pattern Combinators

The core set of combinators is implemented as methods in the class `Pattern`. They can be invoked on any instance of `Pattern` and its subclasses. They can be specialized by any subclasses like any regular method. However, specializing combinator semantics or extending the set of combinators should not violate the pattern matching interface to enable composition.

Application Combinator The application combinator $\Rightarrow$ is implemented as a method named `=>` in the class `Pattern`. Listing 9 shows the implementation of the application combinator.

```
"Pattern>>"
=> closure = (
  ↑ PatternApplication of: self and: closure.
)
```

Listing 9: Implementation of the Application Combinator

The class `PatternApplication` is a subclass of `Pattern` and used to represent a pattern that is the result of invoking the application combinator. The class is also nested in the class `Pattern`. The specialized pattern interface is shown in Listing 10. The slots `pattern` and `closure` are populated upon object initialization, their respective values correspond to the arguments to the primary factory method shown in Listing 9.

```
"PatternApplication>>"
doesMatch: subj else: fail = (
  | bind | bind:: pattern doesMatch: subj else: [ MatchFailure ].
  ↑ bind == MatchFailure
     ifTrue: [ matchFailedFor: subj escape: fail ]
     ifFalse: [ | result proxy |
                proxy:: ProxyReceiver wrap: closure home receiver with: bind.
                closure home receiver: proxy.
                result:: closure valueWithPossibleArgument: bind.
                Binding for: result.
     ]
)
```

Listing 10: Pattern Interface of `PatternApplication`

Lines 8 and 9 of Listing 10 modify the receiver of the closure: We insert a proxy receiver that sends messages to the binding before they are sent to the original receiver. This allows for conveniently accessing the `boundValue` slot of a binding. Line 10 evaluates the closure, optionally passing in the binding as an explicit argument.

```
fib: n = (
  <1> => [ ↑ 0 ] doesMatch: n else: [].
  <2> => [ ↑ 1 ] doesMatch: n else: [ ↑ (fib: n-2) + (fib: n-1) ].
)
```

Listing 11: Example Illustrating Use of the Application Combinator

The example in Listing 11 illustrates the usage of the application combinator. It shows the implementation of a method that returns the Fibonacci numbers. The failure handler is used in the last line of Listing 11 to define the rest of the Fibonacci number sequence.

Disjunction Combinator The disjunction combinator $\cup$ is implemented as a method named `|` in the class `Pattern`. The details are analogous to the implementation of the application combinator and presented in Listing 12. The pattern interface implementation is straight-forward: If the given object matches the first pattern (the receiver of the `|` message), then the respective binding is returned. We use the failure handler to match the subject on the second pattern if the first pattern does not match the given subject.

```
"Pattern>>"
| alternativePattern = (
  ↑ PatternDisjunction of: self and: alternativePattern.
)

"PatternDisjunction>>"
doesMatch: subj else: fail = (
  ↑ pattern doesMatch: subj else: [
      alternativePattern doesMatch: subj else: fail
    ]
)
```

Listing 12: Implementation of the Disjunction Combinator

The example in Listing 13 illustrates the use of the disjunction combinator. We use the combinator to produce a composite pattern for identifying either `1` or `2` and use the application combinator to associate a closure that returns the respective Fibonacci number.

```
fib: n = (
  (<1> | <2>)  => [↑ n-1] doesMatch: n else: [ ↑ (fib: n-2) + (fib: n-1) ]
)
```

Listing 13: Example Illustrating Use of the Disjunction Combinator

Sequence Combinator The sequence combinator ⋗ is implemented as a method named `>>` in `Pattern`. The details are presented in Listing 14. We use the class `PatternSequence` to represent the result of invoking the sequence combinator on a pattern. The respective pattern interface implementation attempts to match the given object on the pattern which was the receiver of the `>>` message. Upon success, the value hold by the `boundValue` slot is passed as a subject to the pattern which was the argument to the sequence combinator.

```
"Pattern>>"
>> rightPattern = (
  ↑ PatternSequence of: self and: rightPattern.
)

"PatternSequence>>"
doesMatch: subj else: fail = (
  | bind |
  bind:: leftPattern doesMatch: subj else: [ MatchFailure ].
  ↑ bind == MatchFailure
      ifTrue: [ matchFailedFor: subj escape: fail ]
      ifFalse: [ rightPattern doesMatch: bind boundValue else: fail ]
)
```

Listing 14: Implementation of the Sequencing Combinator

We introduce the `Minus` pattern in Listing 15 in order to illustrate the use of the sequencing combinator. It is loosely based on *n+k* patterns: It allows for subtracting a number while matching. However, for the sake of simplicity, the pattern neither performs type checks nor verifies the result. It subtracts a given number and wraps the result in a binding instance.

```
Minus k: sub = Pattern ( | subtrahend = sub. | )(
  doesMatch: subj else: fail = (
    ↑ Binding for: subj - subtrahend.
  )
)
```

Listing 15: `Minus` Example

The `Minus` pattern is used in the example in Listing 16. The implementation of `fib:` is equivalent to that in Listing 13, except that the instance of the `Minus` pattern handles the subtraction.

```
fib: n = (
  ((<1> | <2>) >> Minus k: 1)  => [ boundValue ]
    doesMatch: n else: [ ↑ (fib: n-2) + (fib: n-1) ]
)
```

Listing 16: Example Illustrating Use of the Sequencing Combinator

Negation Combinator The negation combinator $\not\leftarrow$ is implemented as a method named `not` in the class `Pattern`. The relevant parts of the implementation are presented in Listings 17 and 18. Instances of `PatternNegation` return a binding indicating success only if the pattern which received the `not` message does not match a given object.

```
"Pattern>>"
not = (
  ↑ PatternNegation of: self.
)
```

Listing 17: Implementation of the Negation Combinator

The example in Listing 19 illustrates the use of the negation combinator. The example reverses the example shown in Listing 13.

```
"PatternNegation>>"
doesMatch: subj else: fail = (
  | bind | bind:: pattern doesMatch: subj else: [ MatchFailure ].
  ↑ MatchFailure == bind
      ifTrue: [ Binding for: subject ]
      ifFalse: [ matchFailedFor: subj escape: fail ]
)
```

Listing 18: Implementation of the Negation Combinator

```
fib: n = (
  (<1> | <2>) not => [ ↑ (fib: n-2) + (fib: n-1) ] doesMatch: n else: [↑ n-1]
)
```

Listing 19: Example Illustrating Use of the Negation Combinator

Conjunction Combinator The conjunction combinator ∩ is implemented as a method named `&` in `Pattern`. The implementation details are very similar to those of the sequence combinator, with the important difference that both patterns are matched on the same object. We provide the details in Listing 20. Line 12 makes the difference to the sequencing operator: The original subject is passed to the second pattern, rather than the object hold by the `boundValue` slot of the binding `bind`.

```
1   "Pattern>>"
2   & rightPattern = (
3     ↑ PatternConjunction of: self and: rightPattern.
4   )
5
6   "PatternConjunction>>"
7   doesMatch: subj else: fail = (
8     | bind |
9     bind:: leftPattern doesMatch: subj else: [ MatchFailure ].
10    ↑ bind == MatchFailure
11        ifTrue: [ matchFailedFor: subj escape: fail ]
12        ifFalse: [ rightPattern doesMatch: subj else: fail ]
13  )
```

Listing 20: Implementation of the Conjunction Combinator

We introduce a simple class hierarchy in Listing 21 to illustrate the use of the conjunction combinator. We define a representation for widgets, where the class `Label` subclasses the class `Widget`. Both classes define an accessor to provide for identification at run-time. Defining accessors of the form `isA` adheres to Newspeak's protocol: Sending `isA` messages to objects, evaluates to `false` if the receiver is the root of the object hierarchy `NewspeakObject`. The method `moveTo:` in the class `Widget` provides for changing the position of a widget in its host container. We omit the implementation for the sake of simplicity of our example.

```
class Widget = ( | color | )(
  isWidget = ( ↑ true )
  moveTo: aPosition = ( ... )
)
class Label = Widget ()( isLabel = ( ↑ true ) )
```

Listing 21: Example Widget Classes

In Listing 22 we describe two patterns that can be used to identify widget objects represented by instances of the classes defined in Listing 21. The first pattern identifies instances which are labels. The second pattern relies on the fact that widgets are colored and attempts to retrieve the color of a given subject and matches it. We omit the implementation of `matchColor:` to concentrate on the pattern interface, however, an implementation based on RGB or HSV values is straight-forward.

```
class LabelPattern = Pattern ()(
  doesMatch: subject else: fail = (
    ↑ subject isLabel
        ifTrue: [ Binding for: subject ]
        ifFalse: [ matchFailedFor: subj escape: fail ]
  )
)
class Colored as: c = Pattern ( | color = c. | )(
  matchColor: subjColor = ( ... )
  doesMatch: subj else: fail = (
    (subj isWidget) ifTrue:[subj color ifNotNil:[ :col | ↑ matchColor: col]].
    ↑ matchFailedFor: subj escape: fail
  )
)
```

Listing 22: Example Patterns for Label Widgets and Color Objects

Listing 23 illustrates several benefits of our pattern matching facilities. The method `moveWidgets:to:` moves a set of widgets to a new position. We use patterns as predicates on widgets, in order to identify patterns of interest. Passing patterns as arguments is possible, because patterns are first class values. An even more succinct version would involve an implementation of `select:` that is parametrized on a pattern, rather than a closure that evaluates to a boolean value. Furthermore, we can compose pattern objects to produce a more accurate pattern by using the conjunction combinator as shown in line 9. The example invocation of `moveWidgets:to:` passes a composed pattern which identifies blue labels.

```
1  moveWidgets: widgetPattern to: aPosition = (
2    allWidgets select: [ :widget |
3      (widgetPattern => [ true ]) doesMatch: w else: [ false ]
4    ] thenDo: [ :widget | widget moveTo: aPosition ]
5  )
6
7  "example use:"
8  moveWidgets: LabelPattern new & (Colored as: Color blue)
9           to: (Position x: 0 y: 0)
```

Listing 23: Example Use of Conjunction Combinator

It is arguable that the definition of patterns for widgets in Listing 22 is rather elaborate. This is due to the fact that we introduce patterns that match arbitrary objects, rather than literals values as shown in previous examples. We aim to address the issue of matching arbitrary objects in Section 4.3 by describing the concept of keyword patterns. Furthermore, we provide literals for keyword patterns as described in Section 4.1. Thus, we offering keyword patterns in order to reduce the overhead that is required for matching objects.

Discussion The class `Pattern` provides for conveniently extending the set of combinators by using the class method `wrap:`. The method expects a closure and returns an instance of the class `Pattern`. The default implementation of `doesMatch:else:` in Pattern relies on a closure stored in the slot `wrappedClosure` that is set correctly by `wrap:`. The method `doesMatch:else:` evaluates the closure with the given object and failure handler as arguments. This implies that the closure should be parametrized over two arguments, analogously to the pattern interface.

```
"Pattern>>"
=> closure = (
↑ Pattern wrap: [ :subject :fail | | bind |
  bind:: doesMatch: subject else: [ MatchFailure ].
  bind == MatchFailure
    ifTrue: [ matchFailedFor: subject escape: fail ]
    ifFalse: [ | result proxy |
               proxy:: ProxyReceiver wrap: closure home receiver with: bind.
               closure home receiver: proxy.
               result:: closure valueWithPossibleArgument: bind.
               Binding for: result.
    ]
  ]
)
```

Listing 24: Alternative Implementation of the Application Combinator

The example in Listing 24 shows an alternative implementation of the application combinator based on `wrap:`. The closure passed to `wrap:` expects two arguments: The object that the pattern is matched on and the failure handler. The closure is implements the same functionality as presented in Listing 9.

4.3 Keyword Patterns

After describing the underlying model used for pattern matching in Newspeak, we proceed to describe an extension based on this model: keyword patterns. They aim to facilitate the matching of arbitrary data structures designed by application developers. Rather than implementing a specific subclass of `Pattern` for each object of interest, keyword patterns rely on a common interface and the overhead needed to enable matching objects is kept at a minimum.

We presented the syntactic production rules for keyword patterns in Section 4.1. They are modeled after keyword messages, which are a unique feature of the Smalltalk family. Keyword messages allow for the description of individual arguments through labels, rather than using the position in a sequence. For example, compare `aDictionary at: #age put: 23` with `map.put("age", 23)`. For keyword patterns, the keywords are used as labels for patterns. Moreover, the sequence of keywords is a pattern by itself, analogously to the selector of a keyword message.

We revisit the example introduced in Section 2 in Listing 3 in order to explain keyword patterns in more detail. We introduce a similar data structure in Listing 25. We define three classes `Num`, `Var` and `Product` that inherit from the class `Term`. The optional message following the class name (for example, `named:` or `of:by:`) introduces the primary factory for each class, which is `new` if the message is omitted. The name following the `=` character is used to explicitly define the superclass. It can be omitted if a class inherits from `NewspeakObject`, which is the root of Newspeak's object hierarchy, as shown for the class `Term`. The content between the vertical bars defines the class' slots as well as their initialization, similar to constructors in other languages. The empty trailing pair of parentheses is used to define the methods of the instance-side mixin of a class (as shown in Listing 26).

```
class Term = ()()
class Num of: n = Term ( | val = n. | )()
class Var named: n = Term ( | name = n. | )()
class Product of: n by: m = Term ( | left = n. right = m. | )()
```

Listing 25: Representing Terms in Newspeak

4.3.1 Enabling Matching on Objects

Matching keyword patterns on objects requires the implementation of a common interface. Each object that aims to support matching a pattern on itself, needs to have an implementation of the method `match:`. The single argument is the keyword pattern that is matched on the object. Consider the source code in Listing 26 that illustrates implementations of `match:`.

```
class Num of: n = Term ( | val = n. | )
( match: pat = ( ↑ pat num: val. ))
class Var named: n = Term ( | name = n. | )
( match: pat = ( ↑ pat var: name. ) )
class Product of: n by: m = Term ( | left = n. right = m. | )
( match: pat = ( ↑ pat multiply: left by: right. ) )
```

Listing 26: Enabling Matching On Algebraic Terms

We augment the data structure presented in Listing 25 by implementations of the `match:` method. Each time, we send a keyword message to the pattern that attempts to identify the object and return the result. A matching keyword pattern corresponds to the keyword message that is sent to the pattern argument. This means that instances of `Num` match the pattern `<num:>` and instances of `Product` match the pattern `<multiply:by:>`.

```
"KeywordPattern>>"
doesMatch: subj else: fail = (
  | res |
  res:: subj match: self.
  ↑ res == MatchFailure
        ifTrue: [ matchFailedFor: subj escape: fail ]
        ifFalse: [ res ]
)
```

Listing 27: Keyword Pattern Interface

Consider the implementation of the pattern interface for keyword patterns shown in Listing 27. It shows how control is passed to the subject by invoking `match:`. This is essential in order to provide for representation independence. The pattern "asks" the object what pattern it aims to match, thus enabling the designer of the data structure to decide which parts of the structure are exposed.

The class `KeywordPattern` leverages the advantages of a dynamic type system and related facilities to enable this correspondence between the message that is sent to the argument pattern and the specified keywords and patterns. Figure 6 aims to illustrate the dialog between a keyword pattern and an object. A simplifier object attempts to match a keyword pattern on an instance of `Produce`. Subsequently, the pattern attempts to match the provided instance of `Product` by sending the `match:` message, passing itself as the single argument. The instance of `Product` "replies" by sending the message `multiply:by:` to the keyword pattern. The message is caught by the `doesNotUnderstand:` facility of the keyword pattern and results in a successful match, which is returned to the simplifier object that tried to match the pattern on the `Product` object.

The fact that Newspeak is dynamically typed, allows for sending arbitrary messages to the pattern. More specifically, there is no need to statically determine whether the pattern is capable of handling the invoked message.

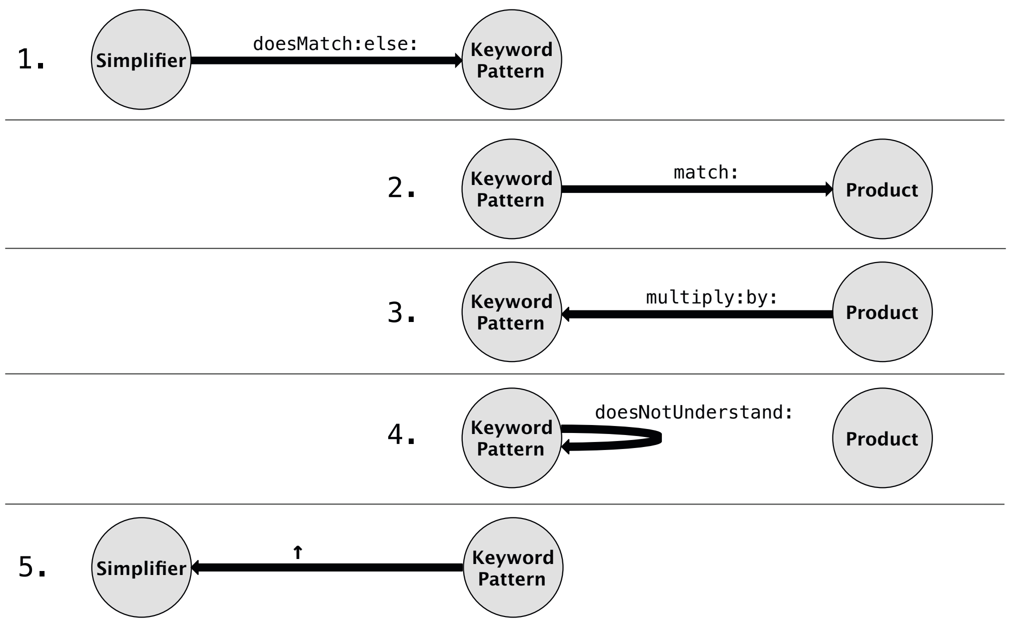


Figure 6: Keyword Patterns

Instead, the pattern employs the `doesNotUnderstand:` facility in order to process messages and attempt to match itself on the message and the provided arguments. Consider the implementation presented in Listing 28.

The keyword pattern is associated with a "selector" analogously to keyword messages in Newspeak. For the pattern `<multiply:by:>` the selector is `multiply:by:`. The implementation in Listing 28 compares the selector of the given message with the one associated with the keyword pattern itself. It indicates a failed match by returning an empty binding (`MatchFailure` evaluates to the value of a slot of the class `KeywordPattern`) or proceeds to match the arguments on the associated nested patterns.

4.3.2 Keyword Bindings

While matching nested patterns on the respective message arguments, the keyword pattern populates a "Keyword Binding" object, which is the result of a successful match. The class `KeywordBinding` subclasses the `Binding` class presented in Section 4.2 and adds functionality to bind values to names that are accessible by sending the name of a value to the binding object. Consider

```
1  "KeywordPattern>>"
2  doesNotUnderstand: msg = (
3    | subj |
4    subj:: thisContext sender receiver.
5    ↑ msg selector = selector
6          ifTrue: [ matchArguments: msg arguments ofSubject: subj ]
7          ifFalse: [ MatchFailure ]
8  )
```

Listing 28: Keyword Pattern Interface Continued I

the implementation of `matchArguments:ofSubject:` in Listing 29, which is invoked in line 6 of Listing 28.

```
1  "KeywordPattern>>"
2  matchArguments: args ofSubject: subj = (
3   | kwBinding |
4    kwBinding:: KeywordBinding for: subj.
5    args with: selectors do: [ :arg :sel | | res |
6            res:: (dict at: sel) doesMatch: arg else: [ ↑ MatchFailure ].
7            kwBinding bind: res boundValue at: sel.
8    ].
9    variables keysAndValuesDo: [ :k :bpat | kwBinding bind: bpat slot at: k ].
10
11   ↑ kwBinding.
12 )
```

Listing 29: Keyword Pattern Interface Continued II

Consider the brief example in Listing 30. The code in lines 5 to 8 of Listing 29 iterates over the message arguments and individual labels in pairs. The slot named `selectors` in `KeywordPattern` holds a collection of the individual labels. The loop would be evaluated only once for the keyword pattern `kwp`. The keyword would be `num` and the argument defined by the message send in the respective `match:` implementation. For the instance `o1` it would be `2` and for `o2` it would be `1` respectively.

Line 6 of Listing 29 attempts to match the argument on the respective pattern. The slot named `dict` in `KeywordPattern` holds a mapping from label

```
kwp = <num: 1>.
o1 = Num of: 2.
o2 = Num of: 1.
```

Listing 30: Keyword Pattern Example

names to respective patterns. In our example based on Listing 30, the mapping would be from `'num'` to `<1>`. Therefore, the matching would succeed on `o2`. Line 7 of Listing 29 binds the resulting value hold by the binding to the keyword's name in the new keyword binding. The resulting value will be accessible by sending `num` to the respective keyword binding instance. Line 9 of Listing 29 binds further values to the new keyword bindings. It is discussed in more detail in Section 4.3.3.

The pattern bound to `kwp` explicitly defines a pattern for the value labeled by the keyword `num`. Because the definition of a sequence of keywords is a pattern in itself, it is is optional to define a pattern on arguments (cf. the grammar extension presented in Listing 5). If omitted, the literal instantiates a wildcard pattern that matches any value. For example, the pattern `<num:>` would match the instances bound to `o1` and `o2`.

We revisit the simplification rule presented in Listing 3 to illustrate the relation between keyword patterns and keyword bindings. The implementation in Listing 31 is equivalent to the Haskell version: It returns the left hand side of a multiplication, if the right hand side is a representation of the number 1 and otherwise the whole expression.

```
simplify: expr = (
  ↑ <multiply: by: <num: 1>> => [ multiply ] doesMatch: expr else: [ expr ].
)
```

Listing 31: Simplifying Algebraic Expressions in Newspeak

The closure argument to the application combinator demonstrates how to access the values bound in the keyword binding. The argument labeled by the keyword `multiply` is accessible inside the closure argument. This is the argument passed to the pattern in the respective `match:` implementation in the last line of Listing 26.

The keyword names are "bound" inside the closure by means of reflection on the closure itself. The keyword binding instance, which is the result of matching a keyword pattern on an object, is entered as a proxy receiver of messages that are sent in the closure argument. This is shown in Listing 10.

Keyword patterns can be nested, as defined by our grammar extensions. This is essential to allow for concise destructuring of nested data structures. The simplification rule makes use of this feature. We use the keyword pattern `<num: 1>` to match the right hand side of the product. The result of matching a keyword pattern is a keyword binding and thus we allow for accessing nested values through keyword bindings. Consider the example in Listing 32. If the given expression is the product of two numbers, we return a representation of the product, otherwise the original expression. The values of the numbers is accessed by sending the `num` message to the keyword bindings, which are the results of matching keyword pattern `<num:>` on the left and right hand side.

```
simplify: expr = (
  ↑ <multiply: <num:> by: <num:>> => [ Num of: multiply num ∗ by num ]
    doesMatch: expr else: [ expr ].
)
```

Listing 32: Accessing Nested Values

Traversing the resulting binding based on keywords is unique to our approach. Traditional pattern matching facilities allow for the use of identifiers similar to variables to bind values to names. Line 9 hints at the processing of pattern variables. We introduce them in the context of keyword patterns and keyword bindings.

4.3.3 Pattern Variables

Patterns variables allow for the labeling of a value with a name. Listing 33 shows a different version of the simplification rule presented in Listing 31 based on pattern variables. We define a pattern variable named `x` in order to refer to the left hand side of the multiplication. We can refer to the value of the variable by sending the name to the keyword pattern. The details for this are presented in Line 9 of Listing 29, where the values of the variables are bound in the keyword binding. More specifically, the slot named `variables` in the class `KeywordPattern` holds a mapping from names to respective values of pattern variables.

```
simplify: expr = (
  ↑ <multiply: ?x by: <num: 1>> => [ x ] doesMatch: expr else: [ expr ].
)
```

Listing 33: Pattern Variables in Keyword Patterns

Pattern variables are implemented as patterns which allows for the uniform definition of the grammar and also the match algorithm in Listing 29. They can be considered as wildcard patterns which lock on the first object. Initially they match any value, similar to names used in traditional pattern matching facilities. However, they store the first object on which they are matched and any subsequent match attempt only succeeds if the objects are identical. We employ a literal pattern to match the passed argument object on any subsequent subjects. This means that the objects are compared by means of `=`. The example in Listing 34 illustrates the use of pattern variables to label and identify a value across different patterns. If the given expression is the product of the same number, then we return the number squared, otherwise the original expression.

```
simplify: expr = (
  ↑ <multiply: <num: ?x> by: <num: ?x>> => [ Num of: x ** 2 ]
    doesMatch: expr else: [ expr ].
)
```

Listing 34: Pattern Variables Across Patterns

The scope of a pattern variable is the outermost enclosing keyword pattern literal. All variables with the same name refer to the same value upon a successful match of the outermost enclosing keyword pattern. As can be seen in the example in Listing 34, the variables are accessible at any of the involved keyword bindings: The message `x` is sent to the keyword binding that is the result of matching the enclosing keyword pattern `<multiply:by:>`, rather than a nested pattern where the variable is actually used. This allows for conveniently labeling values in nested structures and accessing them without traversing the keyword bindings.

4.3.4 Discussion

We describe the common interface for enabling the matching of keyword patterns on objects: An implementation of the `match:` method. We leave the implementation of this method to the implementor of new classes. However, it would be possible to add a default implementation in `NewspeakObject` to accommodate many scenarios. For example, consider offering an implementation that uses introspection to identify the slots of a class. The implementation could send a keyword message to the pattern argument that consists of the names of all slots and their respective values, or even the power set thereof (minus the empty set). However, this would expose the values of an object. To overcome this issue, one might define a common interface for accessing a set of unary messages which could be used for keyword messages in a default `match:` implementation. This would leave it to the implementor of a class to explicitly choose to expose values.

We argue that such a default implementation should be offered by the designer of an object hierarchy, rather than in `NewspeakObject`. Adding an implementation to the root of the object hierarchy should be considered carefully. The application developer has more detailed knowledge about the application domain and thus can determine which information should be exposed or what additional interface should be imposed on subclasses.

4.4 Match Construct

Most languages with pattern matching facilities provide language constructs to enable matching patterns on values. Haskell introduces a `case of` construct and Scala uses `match` for similar purposes. Extending the compiler by a construct for pattern matching enables optimization (for example, identifying redundant cases in a statically typed language [Emi07]) and allows to abandon parentheses for specifying an order of evaluation. However, only developers who are familiar with and have access to the language's compiler are able to modify or extend the respective construct.

We provide a matching construct that is implemented as the method `case:otherwise:`. This is in accordance with the mentality of the host language which implements conditional expressions by means of message sends and closures. It provides for adaption and specialization by application developers. We argue that such flexibility facilitates future changes, while extending a compiler is accessible only to a restricted set of language users.

The method is added to `NewspeakObject`, which is currently the root of Newspeak's object hierarchy. This allows for convenient usage of this construct on all objects. The following listing presents the implementation:

```
"NewspeakObject>>"
case: pattern otherwise: fail = (
  | bind |
  bind:: pattern doesMatch: self else: [ Binding new ].
  ↑ bind isBound
      ifTrue: [ bind boundValue ]
      ifFalse: [ pattern matchFailedFor: self escape: fail ].
)
```

Listing 35: Match Construct

The first argument is the pattern object which is matched on the receiver of the message. The second parameter is a closure which is evaluated if the matching failed. This corresponds to the pattern interface `doesMatch:else:`, which also requires an explicit handling of the failure case. For the match construct, this ensures that application developers are reminded to deal with non-exhaustive patterns, but also provides for a common idiom where the pattern for the last clause is a wildcard (similar to an `else` expression).

```
simplify: expr = (
  ↑ expr case: <multiply: ?x by: <num: 1>> => [ x ]
         otherwise: [ expr ].
)
```

Listing 36: Simplification Using the Match Construct

Listing 36 illustrates the application of the new construct. The presented version closely resembles the Haskell version in Listing 3, without required changes to the compiler for the `case:otherwise:` construct. The only additions to the compiler are the literals for patterns, in order to reduce the required overhead. However, pattern literals are translated to message sends. Meaning they could also be instantiated programmatically and more importantly, can be subject to future adaption by application developers.

⁂

We presented our work to add pattern matching facilities to the Newspeak programming language. We introduced our realization of the model presented in Section 3 and our additions to facilitate concise pattern matching statements: Literals for patterns. We described the concept of keyword patterns that provide for an arguably straight-forward way of enabling pattern matching for regular objects, while preserving representation independence. The following section presents our work for integrating the augmented compiler with the existing Newspeak programming platform, in order to make all our extensions accessible to application developers.

5 Application Developer Tool Support

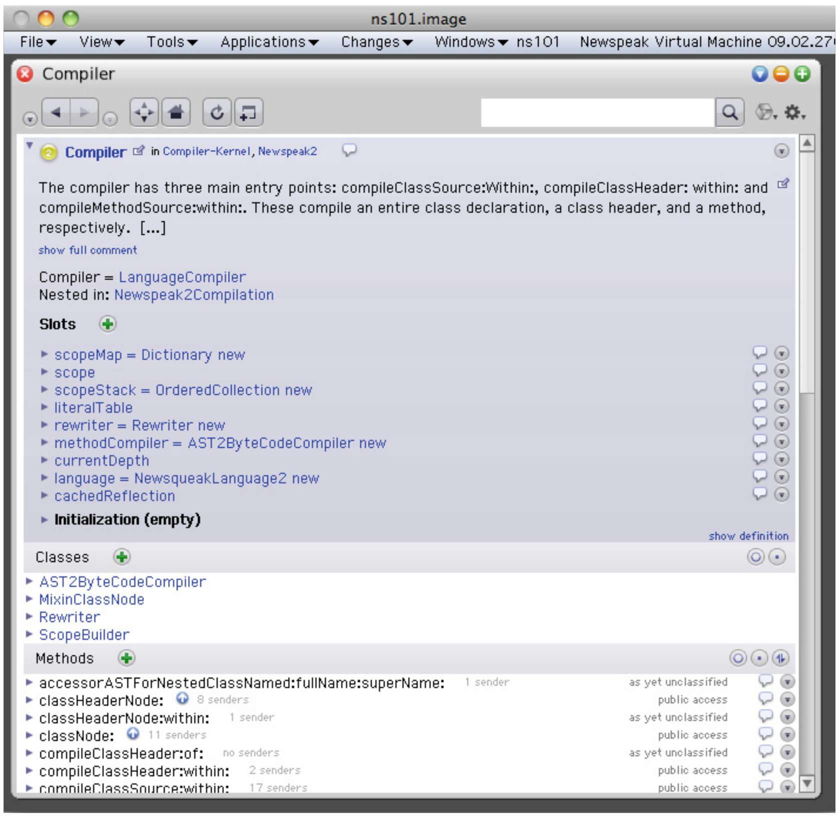
To make our modifications accessible to application developers and allow them to leverage existing development tools, we need to integrate the augmented compiler with the existing programming environment. In this section, we first provide a brief overview of the existing programming environment for the Newspeak programming language. We focus on the aspects that are relevant to our integration, rather than presenting a comprehensive introduction. We discuss our modifications to the environment in order to integrate the augmented compiler framework and describe how application developers can use the new language features.

5.1 The Newspeak Programming Platform

The Newspeak programming platform [BAB+08] encompasses a variety of tools to support application developers: A class browser, debugger, integrated source control management, etc. The class browser follows a document metaphor [Byk08] and is modeled after a web browser. Figure 7 shows the class browser displaying the `Compiler` class, which we augmented to support pattern literals.

5.1.1 Newspeak Integration in Squeak Smalltalk

The current implementation of the Newspeak programming platform is based on Squeak Smalltalk [IKM+97]. This allows application developers to fall back to Squeak Smalltalk's set of tools but also means that multiple languages need to be supported in a single programming environment. Currently, there is support for Smalltalk and different versions of Newspeak. Figure 7 shows a classes which is implemented in the second version of Newspeak, as indicated by the little icon in the top left. The different Newspeak languages represent different development stages of support for the features defined in the language specification [Bra09]. In order to provide for the co-existence of Smalltalk, Newspeak and future language versions, the Newspeak team designed a framework that facilitates the installation of different languages into the Squeak Smalltalk environment: The package `NsMultilanguage`. It provides multiple classes for handling different languages in the same environment, most importantly:



Compiler in Compiler-Kernel, Newspeak2

The compiler has three main entry points: compileClassSource:Within:, compileClassHeader: within: and compileMethodSource:within:. These compile an entire class declaration, a class header, and a method, respectively. [...]

show full comment

Compiler = LanguageCompiler
Nested in: Newspeak2Compilation

Slots

```
scopeMap = Dictionary new
scope
scopeStack = OrderedCollection new
literalTable
rewriter = Rewriter new
methodCompiler = AST2ByteCodeCompiler new
currentDepth
language = NewsqueakLanguage2 new
cachedReflection
```

Initialization (empty)

show definition

Classes

```
AST2ByteCodeCompiler
MixinClassNode
Rewriter
ScopeBuilder
```

Methods

```
accessorASTForNestedClassNamed:fullName:superName:   1 sender      as yet unclassified
classHeaderNode:   8 senders                                       public access
classHeaderNode:within:   1 sender                                 as yet unclassified
classNode:   11 senders                                            public access
compileClassHeader:of:   no senders                                as yet unclassified
compileClassHeader:within:   2 senders                             public access
compileClassSource:within:   17 senders                            public access
```

Figure 7: The Newspeak Class Browser

`Language` This class is the main entry point for the environment to access tools such as a parser and compiler for a particular language. It is an abstract class and should be subclassed when installing a new language, and the respective methods for accessing language specific facilities should be specialized. There are currently subclasses for Smalltalk and different versions of Newspeak. For example, when adding a method to a Smalltalk class `C`, the Newspeak IDE accesses the respective compiler via an instance of a `SmalltalkLanguage`. The class `SmalltalkLanguage` is a subclass of `Language` and associated with the class `C`.

`LanguageCompiler` This class is a wrapper for Squeak Smalltalk's `Compiler` class. It allows for compilers with different interfaces to be used by the Squeak Smalltalk environment for purposes such as compilation of methods or classes.

Its subclass `NS2Compiler` is currently the default compiler for the Newspeak programming platform and it is the goal of our work to provide a substitute which supports pattern matching facilities.

The package also includes classes which deal with reading textual representations of classes into the image and writing them back to the filesystem: `LanguageFileWriter` and `LanguageFileReader`. These textual representations are called "compilation units" and include a definition of the used language. The reader relays most work to the compiler of the respective language, such as an instance of `NS2Compiler`. The writer serializes an existing class definition in the image into a textual representation and is used, e.g., for source control management: The programming platform continuously publishes modified class definitions as text files to a local repository.

The platform enables the Newspeak and Smalltalk languages to coexist, hiding implementation details from application developers. However, the host environment is based on Smalltalk. This means that Newspeak entities eventually need to be translated to Smalltalk. Because the current version of Newspeak is relies on an image which includes the programming environment, this is not only a matter of translating Newspeak code to its Smalltalk equivalent.

In order to leverage the host environment, Newspeak's mixins [BvdAB^{+}10] are represented as Smalltalk classes and their applications (classes) are managed separately. Furthermore, the current Newspeak environment uses Squeak Smalltalk's global namespace represented by the class `Smalltalk` to register top-level Newspeak classes. The registry is used by tools such as the class browser to retrieve information about classes.

5.1.2 Reflection in Newspeak

Newspeak aims to support reflection by means of mirrors [BU04] and provides a mirror library for this purpose. Mirrors are used for introspection on entities such as methods but also provide for dynamically modifying existing classes. The Newspeak programming platform employs mirrors to trigger the compilation and installation of new entities. For example, if an application developer adds a new method `m` to a class `C`, the class browser creates a mirror for `m` and adds it to a mirror for class `C`.

5.2 Installing a Compiler With Pattern Matching Support

We have briefly introduced the Newspeak programming platform focusing on the aspects which are relevant to our goal: Integrating a compiler with pattern matching support. We introduce our additions to enable the installation of the new compiler. We describe our modifications to integrate it with the described system and show how the extended development tools can be used by application developers.

5.2.1 Compiler Integration

We provide a `CompilerIntegration` module which includes the functionality necessary to extend the multi-language framework of the Newspeak programming environment. This includes specialized versions of the classes `Language` and `LanguageCompiler` aiming to integrate the augmented compiler with support for pattern literals.

The new language that is introduced by our integration module is represented by the class `NewspeakLanguage3`. The default language for the current Newspeak environment is Newspeak2 (NS2). Consequently, we label the language with support for pattern literals: Newspeak3 (NS3). We describe the individual parts of the `CompilerIntegration` module that enable the integration of NS3 in more detail:

`NewspeakLanguage3` This class is the specialized version of the `Language` class described above. It is used as an entry point by the environment to access tools that are associated with a specific language. More specifically, `NewspeakLanguage3` provides access to the compiler with support for pattern literals and the respective parser based on the `Newspeak2Parsing` module.

`Newspeak3CompilerAdaptor` This class subclasses the `LanguageCompiler` class and acts as an adaptor to the compiler with support for pattern literals. It bridges the interface required of a compiler by the Squeak Smalltalk environment and the compiler based on the `Newspeak2Compilation` module. It enables the environment to compile and install new top-level classes as well as nested classes using the new compiler framework.

An important difference to the existing compiler `NS2Compiler` is the separation of compilation and installation into the environment. While the previous version compiles classes and immediately installs them in the global namespace represented by the class `Smalltalk`, the new compiler aims for a cleaner interface. The result of a compilation is a mirror of the new class or method that is not yet known to the environment.

In order to install new classes, we use the `AtomicInstaller` module that provides for reflective changes of the environment as well as new installations. The installer expects a nested data structure which represents a mixin and nested mixins, where a single mixin is represented by a mirror. For top-level classes this means that we pass the compilation result directly to the installer, for nested classes we need to recreate the data structure for the enclosing class in order to recreate synthetic methods for accessing nested classes.

`WrappingNS3Parser` This class provides access to the parser associated with the new compiler. It specializes the respective NS2 version in order to emulate the interface of a parser as required by the Squeak Smalltalk environment. Most importantly, it provides for parsing a method and correctly setting the scope.

`NS3SqueakASTBuilder` This class provides functionality to convert an abstract syntax tree (AST) for a Newspeak node into an equivalent Squeak Smalltalk AST representation. This is functionality required by the Squeak Smalltalk environment. We provide support in order to enable the use of the tools provided by the environment, such as the debugger.

`NS3FileWriter` This class specializes the respective version for NS2 to enable the transition from using the global namespace `Smalltalk` to use the generic `SystemMetadata` registry. We describe this issue in more detail in the following Section 5.2.2.

Our modifications to the existing language are literals for patterns in order to reduce the overhead of employing our pattern matching facilities. Because the literals are an isolated addition, we are able to reuse existing functionality for supporting NS2 by subclassing the respective classes. For example the classes `NS3FileWriter` and `NS3SqueakASTBuilder` are subclassing the respective NS2 classes.

In Section 4, we outline our use of the `Newspeak2Compilation` module to implement support for pattern literals. Even though our module aims to integrate the compiler with support for pattern literals, it is generally applicable to a compiler which is based on the `Newspeak2Compilation` module. This allows for future language experiments based on the new compiler framework, not just the integration of pattern matching facilities.

5.2.2 Installing the New Newspeak Language

We described our module for integrating a new compiler into the Newspeak programming environment. We provide this module to enable application developers to use the new language features. We add support to the existing class browser for the new version of Newspeak introduced by the integration module. Furthermore, we modify the environment to support the new class installation strategy employed by our module.

Newspeak Class Browser Applications are developed in the class browser of the Newspeak platform. The browser is able to display classes of multiple languages: Smalltalk and different versions of Newspeak. While NS2 is the default language for the current Newspeak environment, application developers are able to convert existing NS2 classes to Newspeak1 classes. Analogously to the backward conversion, we extend the graphical user interface by an option to convert NS2 classes forward to NS3. Figure 8 shows the menu entry for converting an existing NS2 class to NS3, thus enabling application developers to employ the new language features.

Converting an existing class to NS3 means to associate the existing class with `NewspeakLanguage3`, the main entry point of our integration module to the new compiler. Thereafter all modifications to the converted class are handled by the new compiler framework and can leverage the new language features.

Ultimately, the goal is to replace the current default compiler `NS2Compiler`. However, given that the Newspeak platform already supports multiple languages, we argue that a gradual transition is more feasible. Furthermore, adding an option for conversion is a non-intrusive way to enable application developers to employ the new language features.

The existing framework for multiple languages can be used to install the new language. The class browser uses a registry[3] to define specific layouts

[3]The registry is represented by the class `LanguageUiDescriptionRegistryForApp`.

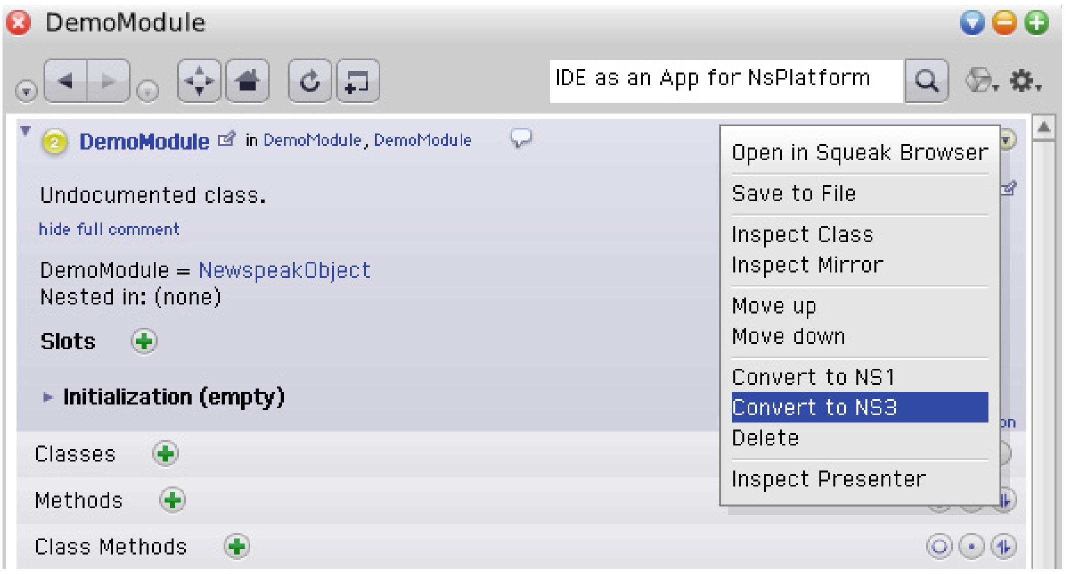


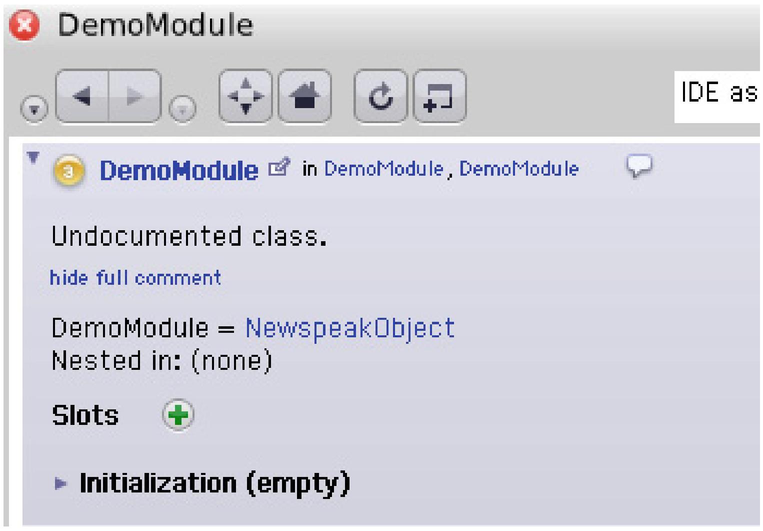
Figure 8: Converting a Class to Newspeak3

for different languages. For example, the layout for Smalltalk has no support for nested classes. We install the mapping for the new Newspeak3 language in order to signal the use of a different compiler. The language version is indicated by the icon in the top left corner of the class browser, as shown on Figure 9.

Installing Classes Into The Environment The Newspeak platform was built gradually on top of the Squeak environment. It is due to this process that the mirror library and parts of the Newspeak environment depend on the global namespace provided by the environment. For example, nested classes are registered in the global namespace with a synthetic name. Several parts of the current environment still rely on this remnant of the evolution of Newspeak.

The `SystemMetadata` class was created to act as a system-wide registry. This provides for a cleaner interface for managing information about classes and replace the `Smalltalk` registry, thus reducing dependencies on the underlying Squeak Smalltalk environment. It is used to register and retrieve information about classes. For example, metadata includes information such as the enclosing class or primary factory of a class.

Our integration module employs the `AtomicInstaller` module in order to



Undocumented class.

DemoModule = NewspeakObject
Nested in: (none)

Figure 9: The Newspeak Class Browser Displaying a Newspeak3 Class

register newly compiled classes. The module installs only top-level classes into the the environment's namespace, dropping the inappropriate use of the global namespace. However, this entailed several modification in the environment where the existing functionality depended on retrieving metadata from the global namespace rather than a proper registry such as `SystemMetadata`. Mostly, these modifications were limited to the mirror library which is used to dynamically modify existing classes.

⁂

We described the Newspeak programming platform and our additions to integrate a new compiler. Our modifications are non-intrusive and enable application developers to leverage the new language features in a comprehensive programming environment. In the following we offer an evaluation of our results.

6 Evaluation

We outlined our motivation and design objectives in Section 2. After presenting the main contributions of our work, we will now revisit the aforementioned design objectives. We evaluate our work by reviewing each objective and highlighting how our design decisions reflect the stated objectives. Moreover, we apply the criteria presented in [EOW07] to our work and discuss the results.

6.1 Design Objectives Revisited

The goal of our work is a smooth extension of the Newspeak programming language by pattern matching facilities. The presented objectives aim to ensure a seamless integration, facilitate future changes and reduce unnecessary overhead for application developers. We return to each individual objective and discuss our results based on the outlined goals.

Enable Abstraction over Patterns Patterns are regular objects in Newspeak. They are first class values like any other object. This allows for flexible use of patterns and facilitates future change as we outlined in Section 2.

```
filter: aList select: pat = (
  ↑ aList select: [ :e | e case: pat => [ true ] otherwise: [ false ] ]
)
```

Listing 37: Generic Filter Implementation

We used a simple example in Listing 4 to illustrate the shortcomings of traditional functional programming languages with respect to abstracting over patterns. The implementation of `filter:select:` in Listing 37 is a simple wrapper of `select:` in order to demonstrate the use of patterns as first class values. It reduces a list to those elements that match the pattern given as an argument. It illustrates the benefits of being able to extract common functionality from different pattern matching constructs (see also, the example for use of the conjunction combinator in Section 4.2.3).

Facilitate Composition Patterns adhere to a well-defined interface that provides for composition: `doesMatch:else:`, which is described in more detail in

Section 4. We provide a set of essential combinators to provide for the composition of different patterns based on this common interface. Application developers can easily extend the existing set of combinators, given that they are implemented as regular methods. For example, consider implementing a combinator that acts as a simple guard to a given pattern:

```
if: guard = (
↑ Pattern wrap: [ :subject :fail |
  guard value
    ifTrue: [ doesMatch: subject else: fail ]
    ifFalse: [ matchFailedFor: subject escape: fail ]
]
)
"example:
  (<var: name> if: [ hasVariables ]) => [ ... ] "
```

Listing 38: Pattern Guard Combinator

The receiver pattern is only matched on a given subject if the block argument evaluates to `true`. This simple example demonstrates the flexibility of patterns in connection with combinators. We believe that making patterns first-class and relying an a simple yet well-defined interface provides for future changes and expressive composition operators.

Preserve Encapsulation Our extension to Newspeak adheres to the principle of data abstraction. We provide keyword patterns which facilitate the matching of regular objects while hiding the concrete implementation. Keyword patterns do not rely on a specific type of an object but instead identify an object by invoking a regular method on the subject. It is left to the developer to implement this method and expose values based on demand.

Enable Ease of Use We provide extensions to facilitate the use of our pattern matching facilities by application developers. We aim to reduce the overhead needed by supporting literals for essential patterns such as keyword patterns. We integrated the augmented compiler into the existing programming environment to make our modifications accessible to application developers. This allows for using existing tools such as a debugger or system browser.

We are able to leverage several features of the host language Newspeak to provide for a seamless integration of our extensions and facilitate the use by application developers. The implementation of keyword patterns and their interface is alleviated by a dynamic type system and the related `doesNotUnderstand:` facility. This provides for labeling values with names rather than encoding everything by the order of elements in a sequence. The reflective capabilities allow for setting bindings as receivers of message sends in closures. This greatly reduces the overhead for accessing values. We aim to accomodate the mentality of late-boundness by translating literals for patterns to message sends. Moreover, the fact that Newspeak supports only message sends at run-time implies that our extension should not rely on a construct such as `case of` in Haskell, which is driven by the compiler.

Exploiting the features of Newspeak ensures that our pattern matching facilities are a smooth and expressive extension to the Newspeak programming language. We implemented a flexible core as a library to the core language. We extended the compiler by literals to facilitate the use of our pattern matching facilities. We believe that our work fulfills to the outlined objectives and proceed to evaluate our results in the context of related techniques.

6.2 Simplifying Algebraic Terms

The work in [EOW07] presents Scala's pattern matching facilities and evaluates them in the context of related techniques. Scala is a pure object-oriented programming language (all values are objects) that aims to incorporate features from functional programming languages, such as pattern matching facilities. The aim of this section is to apply the introduced criteria to our results and contrast Scala's features with our extensions to Newspeak. Please consider [OAC$^+$09, Emi07, EOW07] for more information on Scala and its pattern matching facilities. While we provide a brief overview of these concepts, it is beyond the scope of this report to introduce them.

The following techniques are considered: regular object-oriented decomposition, the visitor pattern, type test and cast, a typecase construct, case classes and extractors. The type-related techniques are not directly relevant in the context of our work, which targets a programming language with a dynamic type system. Case classes and extractors are introduced in Scala to enable convenient pattern matching while preserving data abstraction. Extractors are closely related to our keyword patterns, where our `match:` method is used analogously to the `unapply` method for extractors. Case classes are syntactic sugar which includes the definition of extractor related methods based on constructor arguments. They are closely related to datatypes known from functional programming languages.

The different techniques are evaluated based on solutions to the same problem: Defining a class hierarchy to represent a simple algebraic expression and implementing a basic simplification rule. We include the solution based on Scala's extractors in Listing 39. Extractors are closest to our work and we use them in order to highlight differences. Lines 2 to 5 define the mentioned class hierarchy for representing simple algebraic expression consisting of numbers, variables and products. The following lines 7 to 18 are used to define three separated extractor objects which are used for pattern matching purposes. The `unapply` method is used for destructuring purposes analogously to the `match:` method used by keyword patterns. The return value is of type `Option[+T]` where `Some[T]` indicates a successful match and `None` is used for failure.[4]

Multiple values are represented as a tuple as shown in Line 17. This is clearly different from our use of keyword patterns which provide for labeling individual values. Our keyword syntax can be considered a strength and we believe using it for patterns is a natural extension to Newspeak. Our implementation of keyword patterns relies on Newspeak's dynamic type system and `doesNotUnderstand:` facility. The patterns would not be a natural extension for a statically typed language without keyword messages, such as Scala.

The simplification rule in lines 22 to 25 of Listing 39 demonstrates the use of extractors for pattern matching. Scala provides a `match` construct which allows for the definition of multiple cases, which associate patterns with functionality. The use of extractors in line 23 is translated by the compiler to invocations of the respective `unapply` method and nested patterns are expanded.

We provide a corresponding solution to the outlined problem in Newspeak using our pattern matching facilities and keyword pattern literals as described in previous sections. The source code is summarized in Listing 40.

The example in Listing 41 shows a different implementation that makes

[4]This is related to Tullsen's [Tul00] approach. He employs the type `Maybe a`, where `Nothing` indicates failure and values of `Just a` a successful match. The types `Maybe a` and `Option[+T]` are alternatives to Java™ 's `null`, where `null` corresponds to `Nothing` and `None` respectively. Scala's and Haskell's type checker can statically verify whether the "`null` case" is handled.

```
// Class hierarchy:
trait Term
class Num(val value : int) extends Term
class Var(val name : String) extends Term
class Mul(val left : Term, val right : Term) extends Term

object Num {
  def apply(value : int) = new Num(value)
  def unapply(n : Num) = Some(n.value)
}
object Var {
  def apply(name : String) = new Var(name)
  def unapply(v : Var) = Some(v.name)
}
object Mul {
  def apply(left : Term, right : Term) = new Mul(left, right)
  def unapply(m : Mul) = Some (m.left, m.right)
}

// Simplification rule:
e match {
   case Mul(x, Num(1)) ⇒ x
   case _ ⇒ e
}
```

Listing 39: Simplification Using Scala's Extractors [EOW07]

use of the disjunction combinator to define multiple cases. It aims to resemble the Scala version in Listing 39, but it is equivalent to the version in Listing 40, except for the signaling of a failed match.

It is worth noting that `case:otherwise:` is a regular method call rather than a keyword in the host language. It does not require any modifications to the language compiler and can be specialized in subclasses. Our extensions to the compiler are solely for the sake of convenience. This is an essential difference between our approach to extending Newspeak and Scala's pattern matching facilities which are part of the language's original design.

Refraining from modifying the language compiler to support a construct for matching means that employing our facilities might require some syntactical overhead. More specifically, parentheses are often required to define the

```
class Term = ()()
class Num of: n = Term ( | val = n. | )
( match: pat = ( ↑ pat num: val. ))
class Var named: n = Term ( | name = n. | )
( match: pat = ( ↑ pat var: name. ) )
class Product of: n by: m = Term ( | left = n. right = m. | )
( match: pat = ( ↑ pat multiply: left by: right. ) )

"simplification rule"
e case: <multiply: ?x by: <num: 1>> => [ x ]
  otherwise: [ e ]
```

Listing 40: Simplification Using Keyword Patterns

```
"simplification rule"
e case:
      (<multiply: ?x by: <num: 1>> => [ x ])
    | (<_> => [ e ])
  otherwise: [ NoMatchError signal ]
```

Listing 41: Simplification Using Keyword Patterns Alternative

order in which patterns are composed. For example, compare the different implementations of the simplification rule in Listings 40 and 41. However, we believe that the gained flexibility prevents complications such as the confusion resulting from Haskell's abundance of pattern matching features.

The work in [EOW07] considers three different topics for evaluating the different listed techniques: Conciseness, Maintainability and Performance. Comparing the performance of different languages is not trivial and we have not introduced optimization to improve our pattern matching facilities. Therefore we evaluate our work based on the criteria for maintainability and conciseness.

The Newspeak programming language provides for a concise definition of the class hierarchy. The notational overhead for enabling pattern matching on a class is arguably low: It is reduced to implementing a single method named `match:` by sending a keyword message to the pattern argument. Nested keyword patterns allow for deep matches. Keyword patterns allow application developers to maintain data abstraction. It is straight-forward to add classes with support for similar or different patterns. Adding new patterns is merely a matter of using literals and objects may match multiple patterns.

We include the evaluation summary presented in [EOW07] in the following table. We add the assessment of our solution presented in Listing 40 in the column labeled “Keyword Patterns” in Table 1. The table is not meant to be an absolute assessment of different techniques. Instead, the aim is to illustrate the benefits of pattern matching facilities in the context of related techniques for a particular problem.

Extractors are closely related to our approach using keyword patterns. As we described above, our `match:` method corresponds to the `unapply` method used for extractors. Both approaches allow to maintain data abstraction and facilitate future extension by patterns or classes. However, the syntactic overhead needed for employing our pattern matching facilities is arguably less than that for extractors but not as minimal as for case classes.

	Object-Oriented Decomposition	Visitor Pattern	Type-Test and Cast	Typecase	Case Classes	Extractors	Keyword Patterns
Conciseness							
framework	-	-	+	+	+	-	o
shallow matches	o	-	-	+	+	+	+
deep matches	-	-	-	o	+	+	+
Maintainability							
representation independence	+	o	-	-	-	+	+
extensibility/variants	-	-	+	+	+	+	+
extensibility/patterns	+	-	-	-	-	+	+

Table 1: Evaluation Summary [EOW07]

We quote the results for the related techniques in order to illustrate the benefits of pattern matching facilities in the context of related techniques. We have evaluated our results with respect to conciseness and maintainability. We summarize the findings of the evaluation in [EOW07] that coincide with our experience .

With object-oriented decomposition, it is possible to preserve encapsulation but the traversal of nested data structures quickly becomes tedious. The visitor pattern allows for a clean separation of data structures and respective functionality, but is arguably hard to maintain when adding classes to the data structure. Type related techniques clearly break the principle of representation independence and should be avoided in object-oriented programming languages. Case classes share the benefits and shortcomings of traditional algebraic datatypes: They allow for very concise definition of data structures but break the concept of encapsulation.

We analyzed our results based on the criteria presented in [EOW07] with respect to related techniques. Our work is closely related to Scala's extractors and consequently share similar benefits, most notably the possibility to maintain representation independence. We believe that keyword patterns are a natural extension to the Newspeak programming language and that they leverage the benefits of keyword syntax for pattern matching. Moreover, it is important to notice that the core of our pattern matching facilities requires no modifications to the language's compiler. We believe that implementing pattern matching facilities without extending the language's core allows for flexible use and adaption by application developers.

⁂

We have revisited the design objectives that were outlined in Section 2. This section highlighted how we adhere to these goals in our extension of the Newspeak programming language by pattern matching facilities. Furthermore, we have evaluated our work with respect to related techniques based on the criteria presented in [EOW07]. Next, we present influences on our work and related approaches to adding support for pattern matching in other programming languages.

7 Related Work

We presented pattern matching for Newspeak and evaluated our work in the context of related techniques. As was discussed in Section 2, there is a long history, as well as current research in this field of programming language theory. We provide a more thorough survey of current research that is related to our work. We use this opportunity to contrast our work with existing systems and identify similarities or influences.

7.1 Functional Programming Languages

The concept of matching patterns on data structures originated in the community of functional programming languages. As a consequence, many mature languages such as Haskell or members of the ML family have comprehensive built-in support. But also more recent functional languages such as Qi [Tar08] or Cyclone [Gro06] employ pattern matching.

For statically typed languages, pattern matching is often used to dynamically dispatch functionality at run-time. However, functional programming languages with a dynamic type system also support pattern matching on values to enable the concise definition of functionality. For example, PLT Scheme[5] provides a form named `match`, which provides for matching general values, including support for destructuring lists. The language Erlang [Arm97] has support for matching values such as messages and leverages its pattern matching facilities to further performance optimizations [GD06].

Pattern matching facilities in functional programming languages often break the concept of data abstraction. This prevents a clean separation of algorithms from underlying data structures. More specifically, patterns are constrained to a particular datatype rather than an abstract interface. Views for Haskell [Wad87] and Standard ML [Oka98] address this issue by extending the language. More recently, the work in [SNM07] extends the F# programming language's pattern matching facilities by "active patterns" with similar goals.

Most functional programming languages fail to provide the necessary means to abstract over patterns. As was highlighted in Section 2, this hinders the development of pattern matching facilities without extending the language itself. The work in [PGPnNn96, FB97, Tul00, Rhi09] illustrates the need for

[5]The programming language PLT Scheme is available online at http://www.plt-scheme.org, last accessed: March 16, 2010.

abstracting over pattern in different functional programming languages. But to the best of our knowledge, the proposed approaches were not incorporated into the respective languages.

An important influence on our work is the work by Tullsen [Tul00] on Haskell's pattern matching facilities with respect to abstraction over patterns. He proposes to make patterns first class citizens: Functions of type "`a→Maybe b`". He presents different combinators and demonstrates the usefulness of being able to abstract over patterns as first class values, e.g., by describing a combinator that provides for matching pairs of patterns on pairs of values concurrently. Implementing a similar operator with our pattern matching facilities for Newspeak is rather straight-forward.

Statically typed programming languages such as Haskell often offer improved feedback based on the application of pattern matching facilities. We plan to leverage our pattern matching facilities in combination with Newspeak's metadata system for improved error checking in the future as is outlined in the next section. More importantly, our work addresses both shortcomings of languages such as Haskell with respect to pattern matching facilities: Patterns are inherently first-class values (objects) in Newspeak and keyword patterns preserve data abstraction while facilitating the use of regular objects in pattern matching constructs.

7.2 Object-Oriented Programming Languages

While pattern matching originates in the functional programming community, its key benefits are also applicable in the context of object-orientation. Emerging programming languages aim to combine the well-established features from functional and object-oriented programming, as we outlined in Section 2. Furthermore, recent research attempts to extend languages such as Java™ with pattern matching facilities or create new languages with inherent support.

For the sake of clarity, we distinguish two different aspects of the concept of pattern matching: Extending method dispatch and providing a construct for matching patterns on objects. Our work aims for the second aspect, for which reason we focus our survey more on related constructs rather than changing method dispatch.

7.2.1 Pattern Matching Constructs

We provide an overview of different attempts at extending object-oriented programming languages or languages whose design includes support for pattern matching construct.

Scala [OAC^{+}09] Scala is an object-oriented and statically typed programming languages targeted at the Java™ Virtual Machine platform. It includes generic support for matching objects based on a `case` construct and the concept of "extractors" [Emi07]. Extractors are closely related to "views" for functional programming languages. They enable pattern matching while preserving data abstraction. The name is descriptive in the sense that an extractor method named `unapply` is used for matching and destructuring a value in a single step (analogously the method `apply` can be used as a constructor). More specifically, the `unapply` method is related to the `match:` method used for Keyword Patterns. It is the common interface of objects used to expose relevant constituents. Furthermore, Scala provides "case classes" which are related to algebraic datatypes, support to match the content of XML [BPSM^{+}08] documents [OAC^{+}09] and includes work on optimizing the performance of pattern matching [Emi05].

The pattern matching construct `case` is implemented as a language extension to the compiler. This enables a more succinct syntax, when compared to our work and the respective application of combinators. However, we believe that implementing pattern matching facilities as a library enables specialization by application developers. For example, the semantics of keyword patterns can be specialized by regular subclassing. Furthermore, we believe that leveraging Newspeak's keyword syntax to label the relevant constituents is more intuitive than squeezing data into tuples where order defines meaning.

Matchete [HNBV08] Matchete is an experimental language extension to the Java™ programming language with pattern matching facilities. It provides a uniform approach to different patterns ranging from regular object deconstructors, similar to Scala's extractors, to decoding TCP/IP packets. Consider the following example which implements a function `mult` that returns the product of all the integers in a given list:

The sample shows the generic construct `match` which is an extended version of the regular `switch` statement for Java™ . It is used for matching values and associating patterns and respective functionality. The base case is given

```
1 int mult(IntList ls) {
2   match (ls) {
3    cons~(0, _): return 0;
4    cons~(int h, IntList t): return h * mult(t);
5   }
6   return 1;
7 }
```

Listing 42: Matchete [HNBV08] Example

in line 6 which is evaluated if none of the patterns matches. The patterns in lines 3 and 4 are deconstructors and show the use of nested patterns as well. Deconstructors are suffixed with a tilde character ~ and implemented similarly to regular methods.

Matchete aims to "[unify] different approaches to pattern matching" through extending the language. The pattern matching facilities of our work are flexible enough to express a similar variety of patterns without extending the language core. Moreover, Matchete is lacking the means to abstract over patterns or to provide for operators akin to our combinators. Deconstructors are similar Scala's extractors and as such less intuitive to use, in our opinion, than the keyword syntax leveraged in our approach.

Thorn [BFN$^+$09] Thorn is the continuation of the work on Matchete as a new programming language targeting the Java™ Virtual Machine platform. It is dynamically typed and aims to facilitate a pluggable type system. It incorporates a flexible pattern matching facility, closely related to Matchete. One addition when compared to Matchete is the possibility to split function definitions. Consider the following example that shows the divided implementation of a a function to sum up the values of a given list:

```
fun sum([]) = 0;
  | sum([x,y...]) = x + sum(y);
```

Listing 43: Thorn [BFN$^+$09] Example

Another aspect is the inclusion of an expressive query system with a rich set of operators. However, the facilities for querying and matching objects are not consolidated.

In contrast to our work, Thorn's pattern matching facilities are not a supplementary extension to the language. They are an integral part of the language's original design. This allows for smoothly combining pattern matching and function definition. While it includes interesting additions to the work presented in Matchete, it is unclear whether patterns can be the result of arbitrary statements. More specifically, the ability to abstract over patterns or create operators which combine patterns is not explained. Our approach aims for a closer integration of the facilities for pattern matching and querying of collections and other sources.

JMatch [LM03, LM05] JMatch extends the Java™ programming language with support for "iterable abstract pattern matching" [LM03]. It incorporates facilities for backtracking to support pattern matching. Backtracking is a concept known from logic programming, most prominently the programming language Prolog [Col90]. JMatch adds support for a backward mode which allows the definition of boolean expressions in order to bind values used for pattern matching. While our work aims for a seamless integration with Newspeak, JMatch's changes to the host language are non-trivial, involving a conversion to continuation-passing style in order to enable backtracking. Adding a backwards mode to an imperative language raises issues with regards to program comprehension.

TOM [MRV03] The TOM compiler aims to extend different languages such as C and Java™ by a pattern matching construct named `%match`. The extensions include constructs for defining terms and patterns on them by means of translation rules to statements of the original language. While the changes are not intrusive, they are also not smoothly integrated in contrast to our work. More specifically, employing translation rules rather than features of the host languages allows to target multiple languages but imposes unnecessary complexity for application developers.

7.2.2 Method Dispatch

Enabling the splitting of function definitions in object-oriented programming languages usually entails a generalization of method dispatch to multiple dispatch. This means that methods are dispatched dynamically based on the specific types of selected arguments at run-time, not just the message receiver's type.

The Common Lisp Object System [BDG+88] is one of the first facilities to provide for multiple dispatch in the context of object-orientation by means of "generic functions". The work on "predicate dispatch" [EKC98, Mil04] aims to generalize the concept of single-dispatch traditionally used in object-oriented programming languages such as Smalltalk. It attempts to combine multiple dispatch with pattern matching based on expressive predicates. The work in [MFRW09] addresses the issue of multiple dispatch in the context of the Java™ programming language. They extend the language by dynamic predicate dispatch of methods. The extensions allow to specify a boolean expression as a predicate which is evaluated at run-time and used to determine an adequate method implementation dynamically.

The work on OOMatch [RL07] also extends the Java™ programming language in order to provide for multiple dispatch. It also includes support for destructuring method arguments based on deconstructors. Please consider the following listing for an incomplete example:

```
abstract class Expr { ... }
class Binop extends Expr {
  deconstructor Binop(Expr e1, Expr e2) {
    e1 = this.e1;
    e2 = this.e2;
    return true;
  }
}
class Plus extends Binop { ... }
class NumConst extends Expr { ... }
// example:
Expr optimize(Plus(Expr e, NumConst(0)) { return e; }
```

Listing 44: OOMatch [RL07] Example

The example illustrates the use of a deconstructor to identify the types of constituent values which provides for the destructuring use in method definitions. It also demonstrates that modifying the method dispatch facilities to support pattern matching in Java™ entails non-trivial changes. Especially determining which method is applicable in the context of polymorphism seems needlessly complex.

As was pointed out before, modifying the method dispatch functionality of Newspeak is not the goal of this work. We believe that the benefits of equational reasoning are not easily transferred. It is not clear whether the splitting of method definitions improves program comprehension in an object-oriented context.

7.3 Pattern Languages

Our work and presented related research focuses on integrated pattern matching facilities within a programming language and the involved paradigm. There exist programming languages which build on patterns as their core building block. These languages use the declaration of rules based on patterns and associated meaning as their main method for expressing functionality. They demonstrate the expressivity and flexibility of pattern matching facilities.

The language π [KM09] leverages this approach to support the syntactic growth of a language. Translation rules to existing patterns enable an incremental evolution of the language, similar to macro systems in the LISP family of programming languages. The programming language OMeta [WP07] generalizes the concepts of Parsing Expression Grammars [For04] to handle arbitrary streams of objects. The language aims to facilitate language experimentation. The work of [JK09, JK06] presents a framework for a pattern calculus based on first class patterns. They offer a prototype implementation of the programming language "bondi", which is based on the presented calculus.

⁂

We surveyed related research on different programming languages and different influences on our work. To the best of our knowledge, no other work aims to extend an existing object-oriented and dynamically typed programming language and involves an integration into the respective programming environment. Exploiting keyword syntax is a unique feature of the Smalltalk family. We believe that keyword patterns offer an intuitive way to identify objects of interest and a clean interface to expose information while preserving encapsulation. After contrasting our work with other pattern matching facilities and programming languages, we summarize our results and describe our plans for future work.

8 Summary and Outlook

We presented our work on extending the Newspeak programming language by pattern matching facilities. In this section, we conclude by summarizing the results of our work. Furthermore, we provide several starting points for what we consider future work. These include changes for better tool support, as well as furthering the language's pattern matching facilities and the ease of use.

This report describes the design and implementation of pattern matching facilities for an object-oriented and dynamically typed programming language: Newspeak. Our work includes the seamless integration of our extensions into an existing programming environment in order to facilitate the use of our additions by application developers.

We describe a simple yet flexible model of the pattern matching concepts. The model allows for a smooth integration into an object-oriented programming language. We provide an implementation of this model and demonstrate its applicability. Keyword patterns provide for an intuitive way to identify objects of interest and expose values while preserving the principles of encapsulation. We extended the language's compiler to support various pattern literals in order to reduce the overhead required for employing our pattern matching facilities. The integration of the augmented compiler into the existing Newspeak programming environment enables the utilization of present tools such as the system browser or debugger.

Additional Pattern Literals The presented language extensions include literals for various patterns such as keyword patterns. We plan to extend the set of literals to increase the practicability of pattern matching facilities. Consider the following example which illustrates the application of a pattern as a regular expression to match an identifier:

```
str case: </ ([a-zA-Z][a-zA-Z0-9]*) /> => [ group: 1 ]
    otherwise: []
```

Listing 45: Regular Expression Pattern Example

A more involved example is presented in the following listing. It illustrates the use of patterns to query the content of a structured document. Employing an existing library for parsing XML [BPSM+08] documents and querying them with XPath [BBC+07] statements should enable the implementation of such a pattern literal.

```
doc case:
      <- //singer[@name eq 'Fischer-Dieskau'] ->
        => [ singer collect: [:s | s age]]
    otherwise: []
```

Listing 46: XPath Pattern Example

We have implemented experimental support for regular expression patterns and believe that integrating patterns for querying the content of XML documents would increase the utility of our pattern matching facilities to the application developer.

Integrated Query Language We have presented keyword patterns which allow for an intuitive way of identifying an object based on the content it chooses to expose. We believe that an interesting next step will be extending these capabilities to other sources than plain objects. LINQ [MBB06] aims for a similar goal: providing a uniform interface for querying for the content of collections. This includes regular collections of the host language, but also tables in databases or XML documents.

It is worth noting that patterns and combinators will allow for programmatically creating queries as first class values, rather than a textual representation stored in a string. Therefore, we believe that querying for the content of XML documents or other structured data can be simplified by extending the pattern matching facilities of the Newspeak programming language. The presented combinators already indicate our goal to provide an integrated query language: They resemble operators commonly known from query languages.

We plan on extending the pattern matching facilities to support queries on database tables and other collections. Another option is the support for querying a knowledge base using patterns, similar to logic programming. We believe that the presented core is flexible enough to enable such extensions and have started work towards an integrated query language.

Debugger Support We have extended the compiler for Newspeak to support pattern literals. As described in Section 4.1, a literal is expanded to a regular method send. This means that the debugger displays the expanded version of the pattern literal rather than the concise literal representation. The pattern literal `<val:<val:>>` is translated to a message send to the `Pattern` class that instantiates the corresponding keyword pattern. The expanded version is shown in Listing 47.

```
self Pattern
       keywords: ((self Array new: 1) at: 1 put: #val: ; yourself)
       patterns: ((self Array new: 1) at: 1 put:
         (self Pattern
                 keywords: ((self Array new: 1) at: 1 put: #val: ; yourself)
                 patterns: ((self Array new: 1)
                              at: 1 put: (Pattern wildcard)))))
```

Listing 47: Expanded Nested Keyword Pattern Literal

The debugger will display the source code shown in Listing 47, rather than what was written by the user: `<val:<val:>>`. This behavior is consistent with the handling of other synthesized code such as literals for tuples. Nevertheless this behavior is not intuitive to the application developer. We believe that an option to expand literals on demand would be more appropriate.

Error Checking Pattern matching allows for improved error checking. Statically typed programming languages like Haskell employ the pattern matching construct to provide feedback if a set of patterns is non-exhaustive. The specification for the Newspeak programming language allows for metadata that can be used to support a pluggable type system, similar to the work in [HDN07]. We believe that such a type system could exploit our pattern matching facilities for improved feedback.

⁂

The contributions of our work are self-contained and are accessible to application developers. Nevertheless we aim to improve the user experience in the future and provide further extensions to advance the pattern matching facilities for the Newspeak programming language.

References

[Arm97] Joe Armstrong. The Development of Erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, New York, NY, USA, 1997. ACM.

[BAB$^+$08] Gilad Bracha, Peter Ahe, Vassili Bykov, Yaron Kashai, and Eliot Miranda. The Newspeak Programming Platform. Technical Report, Cadence Design Systems, May 2008.

[BBC$^+$07] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. *W3C Recommendation*, 2007.

[BDG$^+$88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. *SIGPLAN Not.*, 23(SI):1–142, 1988.

[BFN$^+$09] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 117–136, New York, NY, USA, 2009. ACM.

[BPSM$^+$08] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0. *W3C Recommendation*, 2008.

[Bra07] Gilad Bracha. Executable Grammars in Newspeak. *Electronic Notes in Theoretical Computer Science*, 193(1):3–18, November 2007.

[Bra09] Gilad Bracha. Newspeak Programming Language Draft Specification Version 0.05. 2009.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design Principles for Meta-Level Facilities of Object-Oriented Programming Languages. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, 2004.

[Bur69] Rod M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41, 1969.

[BvdAB$^+$10] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as Objects in Newspeak. In *To appear in the Proceedings of the 24th European Conference on Object Oriented Programming, Maribor, Slovenia, June 21-25 2010*, Lecture Notes in Computer Science. Springer-Verlag, June 2010.

[Byk08] Vassili Bykov. Hopscotch: Towards User Interface Composition. In *1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1)*, 2008.

[Col90] Alain Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. *Lecture Notes in Computer Science*, 1445:186–211, 1998.

[Emi05] Burak Emir. Compiling regular patterns to sequential machines. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1385–1389, New York, NY, USA, 2005. ACM.

[Emi07] Burak Emir. *Object-Oriented Pattern Matching*. PhD thesis, École Polytechnique Fédérale de Lausanne, October 2007.

[EOW07] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In E. Ernst, editor, *ECOOP 2007, LNCS 4609*, pages 273–298, Berlin Heidelberg, 2007. Springer-Verlag.

[FB97] Manuel Fähndrich and John Boyland. Statically checkable pattern abstractions. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 75–84, New York, NY, USA, 1997. ACM.

[For04] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM.

[GD06] Qiang Guo and John Derrick. Eliminating Overlapping of Pattern Matching when Verifying Erlang Programs in μCRL. In *12th International Erlang User Conference (EUC'06), Stockholm, Sweden*, September 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Reading, MA, 1995.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[Gro06] Dan Grossman. Quantified Types in an Imperative Language. *ACM Transactions on Programming Languages and Systems*, 28(3):429–475, 2006.

[HDN07] Niklaus Haldiman, Marcus Denker, and Oscar Nierstrasz. Practical, pluggable types. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 183–204, New York, NY, USA, 2007. ACM.

[HNBV08] Martin Hirzel, Nathaniel Nystrom, Bard Bloom, and Jan Vitek. Matchete: Paths Through the Pattern Matching Jungle. *Lecture Notes in Computer Science*, 4902:150, 2008.

[IKM$^+$97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326, New York, NY, USA, 1997. ACM.

[JK06] Barry Jay and Delia Kesner. Pure Pattern Calculus. *Lecture Notes in Computer Science*, 3924:100, 2006.

[JK09] Barry Jay and Delia Kesner. First-class patterns. *Journal of Functional Programming*, 19(02):191–225, 2009.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries The Revised Report*. Cambridge University Press, 2003.

[KM09] Roman Knöll and Mira Mezini. π: a pattern language. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 503–522, New York, NY, USA, 2009. ACM.

[Ler98] Xavier Leroy. The OCaml Programming Language. Online: http://caml.inria.fr/ocaml/index.en.html, 1998.

[LM03] Jed Liu and Andrew C. Myers. JMatch: Iterable Abstract Pattern Matching for Java. In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 110–127, London, UK, 2003. Springer-Verlag.

[LM05] Jed Liu and Andrew C. Myers. JMatch: Java plus Pattern Matching. *Technical Report TR2002-1878, Computer Science Department, Cornell University, October 2002. Software release at http://www. cs. cornell. edu/projects/jmatch*, revised April 2005.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA, 2006. ACM.

[MFRW09] Todd D. Millstein, Christopher Frost, Jason Ryder, and Alessandro Warth. Expressive and modular predicate dispatch for java. *ACM Transactions on Programming Languages and Systems*, 31(2), February 2009.

[Mil04] Todd Millstein. Practical predicate dispatch. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2004. ACM.

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. *Lecture Notes in Computer Science*, pages 61–76, 2003.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[OAC$^+$09] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The Scala Language Specification. *Programming Methods Laboratory, EPFL. Version*, 2.7, 2009.

[Oka98] Chris Okasaki. Views for Standard ML. In *SIGPLAN Workshop on ML*, pages 14–23, 1998.

[PGPnNn96] Pedro Palao Gostanza, Ricardo Peña, and Manuel Núñez. A new look at pattern matching in abstract data types. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 110–121, New York, NY, USA, 1996. ACM.

[Rhi09] Morten Rhiger. Type-Safe Pattern Combinators. *Journal of Functional Programming*, 19(2):145–156, 2009.

[RL07] Adam Richard and Ondrej Lhotak. OOMatch: pattern matching as dispatch in Java. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 771–772, New York, NY, USA, 2007. ACM.

[SDF$^+$09] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robert Bruce Findler, and Jacob Matthews. Revised^6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009.

[SM] Don Syme and James Margetson. The F# Programming Language. *http://msdn.microsoft.com/en-us/fsharp/default.aspx*.

[SNM07] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 29–40, New York, NY, USA, 2007. ACM.

[Tar08] Mark Tarver. The Qi II Programming Language. Online at: http://www.lambdassociates.org/, November 2008.

[Tul00] Mark Tullsen. First class patterns. In E Pontelli and V Santos Costa, editors, *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000*, volume 1753 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, January 2000.

[Wad87] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313, New York, NY, USA, 1987. ACM.

[WP07] Alessandro Warth and Ian Piumarta. OMeta: an Object-Oriented Language for Pattern Matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM.

A Example: Term Simplification in Java™

```
public class Term {
  boolean isProduct() { return false; }
  boolean isVariable() { return false; }
  boolean isNumber() { return false; }
  Term simplify() {
    if(isProduct()) {
      Product p = (Product) this;
      if (p.getRHS().isNumber()) {
        Number n = (Number) p.getRHS();
        if (n.getValue().equals(1)) {
          return p.getLHS();
        }
      }
    }
    return this;
  }
}
class Product extends Term {
  Term lhs, rhs;
  Product(Term lhs, Term rhs) { this.lhs = lhs; this.rhs = rhs; }
  Term getLHS() { return this.lhs; }
  Term getRHS() { return this.rhs; }
  boolean isProduct() { return true; }
}
class Variable extends Term {
  String name;
  Variable(String name) { this.name = name; }
  String getName() { return this.name; }
  boolean isVariable() { return true; }
}
class Number extends Term {
  Integer value;
  Number(Integer num) { this.value = num; }
  Integer getValue() { return this.value; }
  boolean isNumber() { return true; }
}
```

Listing 48: Example: Term Simplification in Java™

ISBN 978-3-86956-065-6
ISSN 1613-5652