Program Fields for Continuous Software

Martin Erwig
School of EECS
Oregon State University
erwig@eecs.oregonstate.edu

Eric Walkingshaw
School of EECS
Oregon State University
walkiner@eecs.oregonstate.edu

ABSTRACT

We propose *program fields*, a formal representation for groups of related programs, as a new abstraction to support future software engineering research in several areas. We will discuss opportunities offered by program fields and research questions that have to be addressed.

1. INTRODUCTION

A *program field* is a set of (syntactically) related programs. A programming language L defines an infinite space of possible programs, and an individual program $P \in L$ occupies a point in this space. In contrast, a program field $\mathcal{P} = \{P_1, \ldots, P_n\}$ corresponds to a connected region or subspace of L.

Program fields open new perspectives on programming artifacts and facilitate new approaches to program development. Like zooming out from a single position in space to a region around it, program fields extend the view from a single program to a program in the context of alternatives with regard to design and implementation.

Program fields change the focus of programming and related activities, including program analysis, testing, and debugging, from a single program to a whole range of related programs that, while sharing certain core parts, can differ in as many respects as needed. Program fields help to keep design options open. They can be effectively employed to delay design decisions and commitments, avoiding their subsequent time-consuming and error-prone reversals when such decisions prove to be premature.

Program fields have the potential to transform the software development process from a discrete, big-step program-to-program hopping approach toward a smoother and more gradual transition between programs via closely related alternatives. The fundamental view of a program changes by adopting the program field perspective. While programs are still conceived as discrete points, they are not isolated but connected to similar programs in their neighborhood. This supports the view of software as belonging to a *continuous* domain.

In this position paper we will illustrate how program fields can effectively address important needs in the area of software engineering. We will argue that an expressive model of program fields provides excellent support for a variety of software engineering tasks and that this should therefore be a focus area of future software engineering research.

After a discussion of the representation of program fields in Section 2, we will present opportunities for their use in Section 3. We will discuss operations on program fields in Section 4 and speculate about a vision of continuous software in Section 5. We end with conclusions in Section 6.

2. REPRESENTING PROGRAM FIELDS

While representations for languages and programs have been studied extensively in Computer Science in the form of grammars and inference systems, abstract syntax trees, or textual representations in concrete syntax, there have been few formal proposals for the systematic representation of groups of programs.

In devising a representation for program fields, it is critical to enable factored representations on different levels of granularity that allow the sharing of common parts. This is important to support the editing and understanding of program fields. For example, on the one hand, an utmost factored representation helps avoid update anomalies when editing a program field. On the other hand, a less factored representation that redundantly repeats common parts supports the idea of showing differences in context. This is already used in tools, such as `diff`, but has proven to be highly effective in other areas as well [3].

The principal idea behind factoring in a program field representation is illustrated in Figure 1: A program field with two programs $\{A\langle P\rangle, A\langle Q\rangle\}$ that share a common part A can be represented by factoring out A to produce $A\langle\{P, Q\}\rangle$.

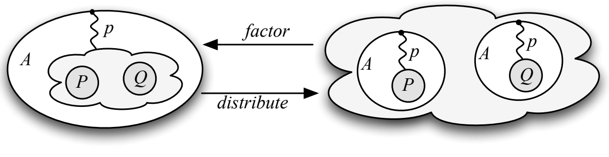
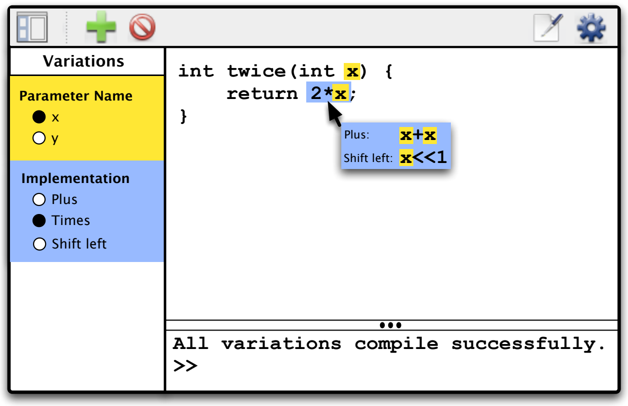


Figure 1: Factoring in program fields.

We have recently developed the choice calculus, a formal representation for software variation [7] that can serve as



```
int twice(int x) {
    return 2*x;
}
```

```
All variations compile successfully.
>>
```

Figure 2: Program field editor.

a basis for representing program fields. A major idea is to organize variation between programs through a concept of *dimensions*, which facilitates the synchronization of differences in disparate parts of programs while still allowing factoring and sharing.

As a simple illustration consider the implementation of a Java method `twice` that returns its argument doubled. The method varies in the name of its parameter (we can have `x` or `y`) and in how it is implemented (we can use addition, multiplication, or bit shifting). A possible choice calculus expression for the resulting program field of six programs is given below. We introduce two dimensions, named *Par* and *Impl*, for the two different kinds of variation. The dimension names are markers for *choices* between alternatives in the code. Dimensions also introduce *tags* that identify their alternatives by name and enable the selection of alternatives.

```
dim Par⟨x, y⟩ in
dim Impl⟨plus, times, shiftLeft⟩ in
let v=Par⟨x, y⟩ in
int twice(int v) {
    return Impl⟨v+v, 2*v, v<<1⟩;
}
```

When selecting one tag in a dimension, say *Par.x*, the alternatives corresponding to x in all choices marked *Par* will be selected. The dimension names tie choices to specific dimensions and ensure their synchronization during selection. This aspect does not apply in this example since the parameter name choice has been shared through a **let** binding, which helps avoid update anomalies when editing program fields. In this example, we can easily change the name of a variable or extend the *Par* dimension by a new alternative without the need to change repeated occurrences of the *Par* choice.

The shown representation offers a rich set of laws and transformations (see [7]) and can be the basis for developing a theory of program fields. However, the notation is probably not suitable for programmers to work with directly. How a more user-friendly presentation could be achieved in an editor is shown in Figure 2. Here choices are colored to indicate their dimension, and a tooltip shows alternatives for a particular choice. The design is similar to the CIDE tool [9].

An intuitive and easy-to-use representation poses many challenges, ranging from the scalability problems, such as how to display many overlapping dimensions, to the hard question of how to specify the extent of editing actions.

For example, suppose we change in the editing window the name `twice` to `double`. Which variations are affected by this change? Is the new name used in all variations, or only in the currently selected one? Another potential interpretation of the change is to create a new variation, which can either be associated with an existing dimension or lead to the introduction of a new one. The design of editing operations and their integration into a user interface are important problems, because an intutive interface is required to make program fields usable in practice.

3. PROGRAM FIELDS IN ACTION

Program fields can support software development in many ways. In the following we highlight some of the most obvious opportunities.

3.1 Programming and Variations

One of the basic tasks of programming is to decide how to represent a specific piece of functionality using the elements of a programming language. In most cases, there is more than one way to do it. The difficulty in making these decisions lies with foreseeing their consequences and impact. Typically, a specific representation makes some tasks easy or efficient while making others more complicated or inefficient. Decisions about the right representations are often difficult since they require anticipation about parts of the programs that are not even written yet.

Program fields can help ameliorate this situation by simply enabling the creation of alternatives. A program that offers two different ways of realizing a specific part of its functionality will be represented by a program field that contains two programs to account for both versions. Extending this view to include choices among several alternatives in many places in the program leads to program fields that share many parts but differ in others.

Once a piece of software has been realized as a program field, it is a simple matter of selecting alternatives to transition from one program to another. By continuously morphing the program field (extending in some places by creating new alternatives and restricting in others) the program field becomes a gradually moving and reshaping window over the infinite program space. Within this window transitions between programs are easy. Moreover, transitions are well defined since they correspond to specific selections among the alternatives offered by the program field.

Finally, being able to see specific program alternatives in the broader context of the whole program field and being able to compare a program part to its alternatives provides a better understanding of software.

3.2 Program Design

Program fields widen the perspective on programming: We are not exclusively focused on one specific solution to a problem, but consider rather a range of related solutions with alternatives that can be beneficial in specific circumstances. Such "solution fields" move implementations closer to the problem. In particular, the possibility for navigation among alternatives can aid understanding of the problem.

Like artists who work with material to express their ideas in the most ingenious way, programmers can mold a program back and forth by creating and moving between alternatives to find an implementation that reflects their best understanding of the problem. The developed alternatives can be helpful to themselves or others to understand the chosen design or to change the design should the context or requirements change. This approach can also be used in the education of software developers to illustrate different ways to realize a particular problem.

Program fields can form the basis for tools that support the generation and exploration of alternative program designs.

More speculatively, program fields might even support the creativity process in generating new solutions. The merging of program fields (see Section 4) could be potentially used to combine different solutions to a problem into one program field, which could then be transformed into a new combined solution. For example, the diet data structure results from combining interval and tree representations for sets [6]. While this data structure was invented from scratch, program field merging could at least have suggested possible solutions for further review and refinement.

3.3 Program Analysis

Program analysis tools discover structures and relationships among program parts to support the understanding of programs by programmers.

One important such tool is `diff`, which is used to understand the differences between two (or more) program versions. Since a factored representation of program fields reveals differences directly and in the context of common program parts, a differencing tool is an immediate, necessary by-product of program fields.

What distinguishes `diff` from most other program analysis tools is that it does not provide information about one program, but rather works on *two* programs, which means that `diff` produces information about a program in the context of another. This idea is at the core of program fields. In a way program fields could be viewed as a systematic representational generalization of `diff`.

Other program analysis tools typically compute specific properties of programs and map (parts of) programs into some (often more abstract) domain. Examples are typing information of identifiers or dependencies between modules. The idea of program fields can be applied to these domains to show, for example, the variation in typing information or module dependencies. Since the fields represent properties of programs, they should be accordingly called *property fields*. Property fields, like program fields, provide a broader perspective on specific program information by presenting it in a contextual fashion.

Finally, program fields suggest the generalization of program analysis to program *field* analysis to answer questions about groups of related programs. Tools such as `diff` are simple examples, but we can envision much more. For example, we can observe that a program field is not an arbitrary subset of L, but rather one whose elements are more or less closely related. Based on notions of program similarity and proximity, we can map out different areas of the program field based on diversity and various measurements of variation.

Property maps generated from program fields may produce something like "geology maps" showing stable areas, "fault lines", etc. that can be exploited by tools for testing or other tasks (see below).

3.4 Testing and Debugging

A program field can represent related programs that all have the same functionality and share, in principle, the same implementations, but differ in fault prevalence and degree of testedness. In such a scenario, program fields can support regression testing [13] by offering a systematic representation of program evolution over a testing and bug-fixing episode.

But even the variation representation offered by program fields can support testing efforts. A program field essentially provides a representation of a software product line [11, 12]. In general, software product lines contain too many products to test each in isolation and require new approaches to spend testing resources more wisely [5]. Since some representations of variations seem to be better suited for efficient testing than others [4], transformation rules that allow the restructuring of program fields can be employed to transform them into a form that is better testable.

The idea behind *goal-directed debugging* [1] is to infer, based on an observed difference between the expected and actual behaviors of a program, sufficient changes to a program that would cause it to show the expected behavior. (This idea is similar to the WhyLine [10], except that the WhyLine only points to program locations and does not infer changes.) Goal-directed debugging can be directly supported by program fields since they offer the ability to explicitly represent code alternatives.

3.5 Summary

Program fields enrich the programming activity by directly supporting ideas from the area of software product lines. Specifically, they support program design through the maintenance of program variations, the smooth transition between programs (adding features, changing behavior, etc.), and the understanding of programs in the context of alternatives.

Moreover, program fields support through their representation various forms of program analysis. Specifically, the presented examples indicate that program fields offer opportunities for the tight integration of tools and analyses with the program representation.

4. OPERATIONS ON PROGRAM FIELDS

Obvious operations on program fields are those that extend or shrink program fields by adding or removing specific variations. While these are important in practice, they seem to be instances of more general operations that should be studied to gain a deeper understanding of program fields and their properties.

The most fundamental operation on program fields seems to be a binary operation to merge program fields. Such an operation can be employed to combine alternative implementations, add program variations, and implement a tool similar to `diff`. The merging of program fields is a combination of union and intersection where common parts of a merged program field are identified and represented as a shared component.

The dual operation "unmerge" (or split) can be used to shrink program fields. When certain parts of a program field become obsolete, unmerge can determine disconnected components that result from removing these obsolete parts.

Looking beyond operations for single, small-step modifications to program fields, we can ask the question of how to evolve program fields systematically. For example, are there scenarios in which scripts or programs for systematic adaptation of program fields could be useful? A related question of language design is whether it is better to keep a separate level of field manipulation operators or integrate operations directly into programming languages, in which case programs could define their own evolution (as is investigated in a related research project [2]).

In addition to modification operations of program fields, there are many interesting questions about program fields that can be answered by defining suitable operations. We can envision measuring the degree of variation within a program field, or correspondingly their cohesion, based on a degree of sharing. Based on such measures, we can also define the notion of distance between two program fields.

More generally, since program fields can represent whole software repositories, we can envision the definition of query languages to find out about programs, program parts, and their relationships, and to explore the represented space of software.

5. CONTINUOUS SOFTWARE

The abstraction of program fields can help shift our understanding of the nature of software as discrete objects to one of a more continuous matter. Remodeling a house is a process that is marked by discrete changes (taking out a wall, replacing a bathtub, etc.). Even though some parts may not function as usual, the house is still mostly usable during the remodeling phase. The whole process can be planned and managed in steps that can ensure in most cases a relatively smooth transition and that lets remodeling be perceived as continuous. Although some research is done in this area [8], the situation for software is typically very different because a minor change to a program might render it completely unusable—it might not even compile, and the transition from one program version to another is typically perceived as a very abrupt and discrete process.

Existing systems for producing software product lines or managing revisions to software over time can be considered special cases of the more general view offered by program fields. While revision control systems enable a relatively continuous view of software in the temporal dimension, variation in other dimensions (via branching) are clumsy and discrete. Software product line systems provide better support for variability in many dimensions, but typically require substantial planning and diligence to use effectively, making them not very conducive to a continuous, evolutionary view of software change.

In many domains we are used to a "continuity of change" principle that lets us expect the nature and behavior of objects to change proportionally with changes applied to them. This is not always the case, but in many everyday situations, minor changes can be managed easily, and big changes can be decomposed into smaller oones.

The vision of *continuous software* is to achieve a similar situation for software, where changes can be gradually phased in without major disruptions. As one way to accomplish this goal, program fields can serve as a representation that fills the empty space between programs that often seems to be an unsurpassable barrier separating them.

6. CONCLUSIONS

Program fields promote *groups of programs* as a fundamental unit of abstraction in software engineering. We have described various applications of program fields and their support for a continuous vision of software.

However, all these benefits and opportunities do not come for free. Program fields are necessarily more complex than single programs and thus require more effort for maintenance. Program fields are also more difficult to edit than single programs. An important scientific question to be addressed will be whether the benefits of program fields are worth the additional costs.

Acknowledgments

This work is supported by the Air Force Office of Scientific Research under the grant FA9550-09-1-0229 and by the National Science Foundation under the grant CCF-0917092.

7. REFERENCES

[1] R. Abraham and M. Erwig. GoalDebug: A Spreadsheet Debugger for End Users. In *29th IEEE Int. Conf. on Software Engineering*, pages 251–260, 2007.

[2] T. Bauer, M. Erwig, A. Fern, and J. Pinto. Adaptation-Based Program Generation in Java. Submitted for Publication.

[3] C. Chambers, M. Erwig, and M. Luckey. SheetDiff: A Tool for Identifying Changes in Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, 2010. To appear.

[4] Myra B. Cohen. Personal communication, 2010.

[5] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *International Symposium on Software Testing and Analysis*, pages 129–139, New York, NY, USA, 2007. ACM.

[6] M. Erwig. Diets for Fat Sets. *Journal of Functional Programming*, 8(6):627–632, 1998.

[7] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Transactions on Software Engineering and Methodology*, 2010. To appear.

[8] M. Hicks and S. Nettles. Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, 2005.

[9] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 311–320, 2008.

[10] Andrew J. Ko and Brad A. Myers. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *30th Int. Conf. on Software Engineering*, pages 301–310, 2008.

[11] D. L. Parnas. On the Design and Development of Program Families. *IEEE Trans. on Software Engineering*, 2(1):1–9, 1976.

[12] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlang, Berlin Heidelberg, 2005.

[13] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering*, 27:929–948, 2001.