Programs are Abstract Data Types

Martin Erwig
Oregon State University
erwig@cs.orst.edu

Abstract

We propose to view programs as abstract data types and to perform program changes by applying well-defined operations on programs. The ADT view of programs goes beyond the approach of syntax-directed editors and proof-editors since it is possible to combine basic update operations into larger update programs that can be stored and reused. It is crucial for the design of update operations and their composition to know which properties they can preserve when they are applied to a program.

In this paper we argue in favor of the abstract data type view of programs, and present a general framework in which different programming languages, update languages, and properties can be studied.

1 Introduction

The largest fraction of software development costs is spent on software maintenance. One important part of software maintenance is the process of updating programs in response to changed requirements. The way in which these updates are performed has a considerable influence on the reliability, efficiency, and costs of this process.

Now a common scenario is the following: a programmer has to perform a couple of changes to a program. She starts her favorite text editor, changes the program, and saves the file. Finally, she tries to recompile the program, but in many cases the compiler will report syntax and type errors. Even if the required program changes are minimal, inconsistencies can be introduced by editing operations quite easily. Even worse, logical errors can be introduced by program updates that perform changes inconsistently. These logical errors are especially dangerous because they might stay in a program undetected for a long time.

The principal problem that causes this unfortunate situation is the low-level view of programs that is revealed by the above "text editor" approach: a program is simply regarded as a sequence of characters, and the operations on programs are basically that of inserting and deleting characters. This view does not at all reflect the structure of programs, and adding or deleting single characters are simply the wrong operations on programs.

Therefore, we propose to view a program as an abstract data type and to perform program changes by well-defined operations offered by this abstract data type. From this point of view it is obvious that operations like

$$addChar : Program \times Pos \times Char \to Program \times Pos$$

are meaningless, or cannot be defined in a reasonable way. Instead, we can well imagine having operations like:

$$\begin{aligned} &addType &&: Program \times Name \times Type \to Program \\ &addFun &&: Program \times Name \times Expr \to Program \\ &dropFunPar &&: Program \times Name \times Var \to Program \\ &\ldots \end{aligned}$$

Performing program updates in a more structured way is actually not a new idea. There exist a couple of program editors that can guarantee syntactic or even type correctness and other properties of changed programs. Examples for such systems are Centaur [2], the synthesizer generator [3], or *CYNTHIA* [5]. The view underlying these tools are either that of syntax trees or, in the case of *CYNTHIA*, proofs in a logical system for type information.

Viewing programs as abstract data types takes the idea of program editors one step further in two respects: first, we are not constrained to one particular representation. Instead, we just have to define a suitable set of sorts, and this offers more freedom and an interesting design space for the definition of program update operators. Second, the definition of basic update operations can (and should) be complemented by combinators to build more complex updates from basic operations. The goal is to be able to write programs that update programs. It is this second aspect that clearly goes beyond the idea of program editors that mainly offer an online "one-operation-at-a-time" mode, and even though some of these operations are of a very high-level nature and actually combine several lower-level operations, there is no way to arbitrarily combine editor operations and save them and later reuse them. Related to our approach is the work by Bjørner who has investigated a simple two-level

lambda calculus that offers constructs to generate and to inspect (by pattern matching) lambda calculus terms [1]. In particular, he describes a type system for dependent types for this language. However, in his approach it is difficult (and in some cases impossible) to break down complex updates into smaller parts.

Update programs are no end in themselves, and this means that we have to restrict update programs to those that guarantee a certain amount of correctness for the programs they produce. One purpose of this paper is to lay out a research plan for this approach, which is to find update languages that are powerful enough to describe reasonably complex program updates and that are at the same time safe enough to guarantee syntactic and type correctness (and possibly other correctness criteria).

Thus, the goal of this paper is *not* to present one update language for one particular programming language preserving one particular set of properties, but rather to explain the general approach that has many different possible applications. In particular, we will address a step that even precedes the definition of ADT operations, namely we investigate conditions that assure that update operations are safe with respect to certain language properties. These insights can then be used in the design of ADT operations, in particular, to judge their safety with regard to these language properties.

The rest of this paper is structured as follows: in Section 2 we describe a general model of program updates. In particular, we describe the abstract data type view of programs and how programming languages and program update languages are related under this view. An important step is to relate properties of update languages to properties of programming languages, and this opens the view on many theorems that are still to be found and proved. In Section 3 we then briefly sketch how an effectively computable property of the update language can guarantee the type correctness of programs. Conclusions given in Section 4 complete this paper.

2 A Generic Model for Program Updates

A general scenario for program updates is given by the following definitions. We consider a language P (called *object language*) and use p to denote programs written in P. A *property* on P is by a function π from P into a suitable domain D. For example, a type system for P is given by a function $\tau : P \to T$, where T is the set of types. Even the semantics of P can be regarded as a property on a domain of semantic values.

Each domain D contains two kinds of values: (i) well-defined values ($D^\diamond$) and (ii) error values (D^ε) so that $D = D^\diamond \cup D^\varepsilon$. This means the fact $\pi(p) \in D^\varepsilon$ indicates a program error with respect to the property π, and $\pi(p) \in D^\diamond$ expresses that p is correct regarding property π. We abbreviate the latter by writing $\pi^\diamond(p)$, that is,

$$\pi^\diamond(p) :\Longleftrightarrow \pi(p) \in D^\diamond$$

In that case we say that p is π-*correct* or π-*valid*.

Updates are given by expressions of an update language U. Updates transform programs, that is, the semantics of updates is a function $[\![\cdot]\!] : U \to (P \to P)$.

Since U is a language, we can identify properties for U (as for P). We use μ and C to denote a property and its domain on U.

Definition 1 (Safety of Updates) *An update u is said to be* safe *with respect to the language property π (or, u is π-safe, for short) if and only if*

$$\pi^\diamond(p) \Longrightarrow \pi^\diamond([\![u]\!](p))$$

Observation: any π-safe update u induces (or corresponds) to a save transition function on the domain D: if $\pi(p) = d$ and $\pi([\![u]\!](p)) = d'$, then the induced update on D is given by $u_D(d) := d'$ where $d, d' \in D^\diamond$ (see Figure 1). In other words, π-correctness is a homomorphism on π-save updates.

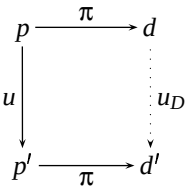

Figure 1. Relationship between updates and properties.

Our goal is to find characterizations of safe updates, that is, we want to find properties μ (and corresponding domains C) such that μ-correctness implies π-safety. This means: given P, π, D, and U, find μ and C such that:

$$\mu^\diamond(u) \Longrightarrow u \text{ is } \pi\text{-save}$$

or:

$$\mu^\diamond(u) \Longrightarrow (\pi^\diamond(p) \Longrightarrow \pi^\diamond([\![u]\!](p)))$$

This generic schema for theorems is illustrated in Figure 2.

To pursue this goal we have to make assumptions about the involved languages and domains. In the remainder of this section we fix some notations that are summarized in Figure 3 that serves as a reference table.

P is usually defined by a grammar, say G, and we will be mainly concerned with P's abstract syntax. On this abstract syntax level a program can be viewed as a term over a signature, and regarding the term view, we consider the signature that is obtained by taking G's nonterminals as sorts and

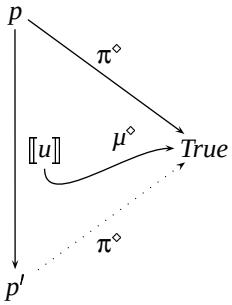

Figure 2. Safe Update Theorems.

having for each production an operation whose type is given by the nonterminals in the production (we omit the formal construction here for brevity). If $\Sigma = \langle S, F \rangle$ is the signature thus obtained, we write $T(\Sigma, W)_s$ to denote the set of terms of sort s where $W = \bigcup_{s \in S} V_s$ is a set of sorted variables. The ground terms of sort s are given by $T(\Sigma)_s := T(\Sigma, \varnothing)_s$, and whenever the signature Σ is clear from the context, we also write more succinctly $T(W)_s$ for $T(\Sigma, W)_s$ and T_s for $T(\Sigma, \varnothing)_s$. An abstract syntax term $f(t_1, \ldots, t_n)$ can also be represented as a tree with node f having the abstract syntax trees for $t_1, \ldots, t_n$ as subtrees.

To relate U and properties on U to those on P, we employ an operational model in which U is a language of term update operations. More specifically, term updates can be expressed by rewrite rules. A rewrite rule for P essentially consists of a pair of patterns (l and r) where a pattern is an abstract syntax term possibly containing variables, that is, $l, r \in T(\Sigma, W)_s$. Complex updates are given by collections of update rules, that is, given the rewrite rules $u_1, \ldots, u_n$, the set $\{u_1, \ldots, u_n\}$ is a valid (complex) update.

A summary of the notation employed so far is given in Figure 3.

P	Object language
Σ	Signature representing the abstract syntax of P
W	Meta-variables for update rules
s	Nonterminal of P and a sort of Σ
U	Update language
$l \rightsquigarrow r$	Update rule, $l, r \in T(\Sigma, W)_s$
$[\![u]\!](p)$	Result of applying update u to program p
$\pi : L \to D$	Object language property on a domain D
$\mu : U \to C$	Update language property on a domain C
D^{ε}	Error values of domain D
$D^{\diamond}$	Well-defined values of domain D
$\pi^{\diamond}(p)$	Well-definedness of program p with respect to π

Figure 3. General model of program updates.

The generic update model defines a framework to study many different languages with many different properties and how these are preserved (or not) under a variety of update operations.

3 Type-Correct Updates

To illustrate the ideas of the previous section, we sketch the development of a criterion for update programs that ensures the preservation of type correctness for updated programs. For simplicity we use the explicitly typed lambda calculus ($\lambda^{\to}$) as an object language.

A principal problem for type-consistent program updates is caused by the fact that the typing information is not a context-free property which means that a single basic update cannot be, in general, type correct. In other words, an update (that affects typing at one place) generally needs one or more further updates at other, remote places to reinstate type correctness.

The general assumption behind our approach is that a complex program update consists of several basic updates $u_1, \ldots, u_n$ that, when applied individually, might introduce type errors (or inconsistencies), but as a whole can work together to preserve type correctness—a complex update is like a transaction that guarantees consistency at the end. The described idea can be formalized in a sequence of steps: first, we define the notions of *type change* and *changes in type assumptions*. Then we define what type changes are induced by update rules, and we divide them into two categories: (R) changes in assumptions that are required to check the types of a rule, and (P) changes in assumptions that are generated or provided during typechecking. Then we can define a "covering" criterion that essentially expresses that

- for any R-change there exists a P-change of which it is an instance, and
- for any P-change there exists an R-change that is an instance of it.

Here, "instance" essentially means the instance relationships of types, but this notion has to be suitably generalized to type changes.

We typecheck both sides of an update rule $u = l \rightsquigarrow r$ individually by a modified type inference algorithm $\mathcal{A}$ (which can actually be considered an assumption inference algorithm) that is more permissive with regard to identifiers for which type assumptions are "missing": whenever the algorithm unsuccessfully tries to find a type for, say f in an assumption Γ, this does not lead to reporting a type error; instead, a most general assumption for f (that is, a type variable a) is generated and added to Γ. We keep track of all generated assumptions in a set Λ that is returned together with the inferred type. Each generated assumption is marked with a tag P or R so that it is known whether a

particular assumption is provided or required.

The algorithm $\mathcal{A}$ is shown in Figure 4. We use the convention that $\hat{a}$ denotes always a new type variable, and that $\mathcal{U}(t,t')$ denotes the most general unifier (mgu) θ for two (type) terms t and t'.

The algorithm is very similar to ordinary type checking/inference algorithms. The main difference is the generated assumptions in the rules $\text{VAR}_{\mathcal{A}}$ and $\text{ABS}_{\mathcal{A}}$. In $\text{VAR}_{\mathcal{A}}$ the second rule captures the situation when the type of a free variable v of a rule pattern is type checked: there is no assumption for v in Γ, and this fact is recorded by generating the required assumption $v \overset{}{\underset{R}{\mapsto}} \hat{a}$. This indicates a potential source of errors, but as long as some other update rule generates a "matching" provided assumption, no type error will be eventually produced. In the rule $\text{ABS}_{\mathcal{A}}$ the explicit typing of the variable v provides a type information that is recorded as $v \overset{}{\underset{P}{\mapsto}} t'$ in Λ. In the rule $\text{APP}_{\mathcal{A}}$ no new information is created, only the generated assumptions of e_1 and e_2 are combined and possibly instantiated by a substitution that results from the unification operation.

$$\text{VAR}_{\mathcal{A}} \quad \frac{\Gamma(v) = t}{\mathcal{A}(\Gamma, v) = (\varnothing, t)} \qquad \frac{\Gamma(v) = \bot}{\mathcal{A}(\Gamma, v) = (\{v \overset{}{\underset{R}{\mapsto}} \hat{a}\}, \hat{a})}$$

$$\text{META}_{\mathcal{A}} \quad \frac{\Gamma(\underline{v}) = t}{\mathcal{A}(\Gamma, \underline{v}) = (\varnothing, t)} \qquad \frac{\Gamma(\underline{v}) = \bot}{\mathcal{A}(\Gamma, \underline{v}) = (\{\underline{v} \overset{}{\underset{R}{\mapsto}} \hat{a}\}, \hat{a})}$$

$$\text{APP}_{\mathcal{A}} \quad \frac{\mathcal{A}(\Gamma, e_1) = (\Lambda, t_1) \qquad \mathcal{A}(\Gamma, e_2) = (\Lambda', t_2) \qquad \mathcal{U}(t_1, t_2 \to \hat{a}) = \theta}{\mathcal{A}(\Gamma, e_1\ e_2) = (\theta(\Lambda.\Lambda'), \theta\hat{a})}$$

$$\text{ABS}_{\mathcal{A}} \quad \frac{\mathcal{A}(\Gamma.\{v \mapsto t'\}, e) = (\Lambda, t)}{\mathcal{A}(\Gamma, \lambda v{:}t'.e) = (\Lambda.\{v \overset{}{\underset{P}{\mapsto}} t'\}, t' \to t)}$$

Figure 4. Assumption inference algorithm.

With the help of $\mathcal{A}$ we can determine the type changes that are introduced by update rules. Next we create a set of "assumption differences" $\Delta(l,r)$ by comparing the inferred assumptions for both parts of a rule. Any such assumption difference is basically given by a variable v and a pair of types, t and t', and means that the type assumption for v has changed from t in l to t' in r. We write this as: $v \overset{}{\underset{P}{\mapsto}} t \rightsquigarrow t'$ (for a provided assumption). Then the assumption differences for a complex update are given by the union of differences for each individual rule: $\Delta(u) = \cup_{l \rightsquigarrow r \in u} \Delta(l,r)$. Based on the assumption differences collected in Δ we can finally define the above sketched covering property on (complex) updates.

With these definitions we can basically show that updates that satisfy the covering criterion preserve type correctness of updated programs.

4 Conclusions and Future Work

We have demonstrated that it is possible to design languages for programming program updates that preserve properties of object programs like syntactic and type correctness. Future work extends the initial proposal in several directions:

1. *Realistic Object Language.* By extending $\lambda^{\to}$ by type inference, polymorphic types, and in particular, data type constructors and pattern matching, we reach a realistic model for languages like Haskell or ML. These extensions are also necessary to be able to investigate high-level operations on programs, such as extending a data type by a new constructor and correspondingly extending pattern matching and constructor applications.
2. *Complete Update Language.* The update language has to be extended by recursion operators and by "congruence operators" [4] that allow to move updates along specific paths of abstract syntax trees. The goal is to provide precise control (on a high-level) of where to apply which updates. Together with other standard extensions, such as conditional rewriting, we can then define high-level language-specific update operators and wrap them into a user-friendly syntax.
3. *Language Properties.* For all extensions we have to investigate what language properties can be preserved under which conditions. Moreover, there are properties of U that deserve attention in their own right. For example, the independence/interference of basic updates, or conditions for updates being composable. This is of particular importance for the development of a library of reusable program updates.

References

[1] N. Bjørner. Type Checking Meta Programs. In *Workshop on Logical Frameworks and Meta-Languages*, 1999.

[2] P. Borras, D. Clèment, T. Despereaux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. In *3rd ACM SIGSOFT Symp. on Software Development Environments*, pages 14–24, 1988.

[3] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989.

[4] E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *3rd ACM Int. Conf. on Functional Programming*, pages 13–26, 1998.

[5] J. Whittle, A. Bundy, and H. Lowe. An Editor for Helping Novices to Learn Standard ML. In *14th Int. Conf. on Automated Software Engineering*, 1999.