Type-Safe Update Programming

Martin Erwig and Deling Ren

Oregon State University
Department of Computer Science
{erwig,rende}@cs.orst.edu

Abstract. Many software maintenance problems are caused by using text editors to change programs. A more systematic and reliable way of performing program updates is to express changes with an update language. In particular, updates should preserve the syntax- and type-correctness of the transformed object programs.
We describe an update calculus that can be used to update lambda-calculus programs. We develop a type system for the update language that infers the possible type changes that can be caused by an update program. We demonstrate that type-safe update programs that fulfill certain structural constraints preserve the type-correctness of lambda terms.

1 Introduction

A major fraction of all programming activities is spent in 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. Text editors are a common tool used to change programs, and this fact causes many problems: for example, it happens quite often that, after having performed only a few minor changes to a correct program, the program consists of syntax and type errors. 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. These facts are not surprising because the "text-editor method" reveals a low-level view of programs, namely that of sequences of characters, and the operation on programs offered by text editors is basically just that of changing characters in the textual program representation.

Alternatively, one can view a program as an element of an abstract data type and program changes as well-defined operations on the program ADT. Together with a set of combinators, these basic update operations can then be used to write arbitrarily complex *update programs*. Update programs can prevent certain kinds of logical errors, for example, those that result from "forgetting" to change some occurrences of an expression. Using string-oriented tools like awk or perl for this purpose is difficult, if not impossible, since the identification of program structure generally requires parsing. Moreover, using text-based tools is generally unsafe since these tools have no information about the languages of the programs to be transformed, which makes the correct treatment of variables impossible because that requires knowledge of the languages' scoping rules. In contrast, a promising opportunity offered by the ADT approach is that effectively checkable criteria can guarantee that update programs preserve properties of object programs to which they are applied; one example is type correctness. Even though type errors can be detected by compilers, type-safe update programs have the advantage that they document the performed changes well. In contrast, performing several corrective updates to a program in response to errors reported by a compiler leaves the performed updates hidden in the resulting changed program.

Generic updates can be collected in libraries that facilitate the reuse of updates and that can serve as a repository for executable software maintenance knowledge. In contrast, with the text-editor approach, each update must be performed on its own. At this point the safety of update programs shows an important advantage: whereas with the text-editor approach the same (or different) errors can be made over and over again, an update program satisfying the safety criteria will preserve the correctness for all object programs to which it applies. In other words, the correctness of an update is established once and for all. One simple, but frequently used update is the safe (that is, capture-free) renaming of variables. Other examples are extending a data type by a new constructor, changing the type of a constructor, or the generalization of functions. In all these cases the update of the definition of an object must be accompanied by corresponding updates to all the uses of the object. Many more examples of generic program updates are given by program refactorings [10] or by all kinds of so-called "cross-cutting" concerns in the fast-growing area of aspect-oriented programming [1], which demonstrates the need for tools and languages to express program changes.

The update calculus presented in this paper can serve as an underlying model to study program updates and as a basis on which update languages can be defined and into which they can be translated.

Our goal is *not* to replace the use of text editors for programming; rather, we would like to complement it: there will always be small or simple changes that can be most easily accomplished by using an editor. Moreover, programmers are used to writing programs with their favorite editor, so we cannot expect that they will instantly switch to a completely new way of performing program updates. However, there are occasions when a tedious task calls for automatic support. We can add safe update programs for frequently used tasks to an editor, for instance, in an additional menu.[1]

Writing update programs, like meta programming, is in general a difficult task—probably more difficult than creating "normal" object programs. The proposed approach does not imply or suggest that every programmer is supposed to *write* update programs. The idea is that update programs are written by a experts and used by a much wider audience of programmers (for example, through

[1] This integration requires resolving a couple of other non-trivial issues, such as how to preserve the layout and comments of the changed program and how to deal with syntactically incorrect programs.

a menu interface for text editors as described above). In other words, the update programming technology can be used by people who do not understand all the details of update programs.

In the next section we illustrate the idea of update programming with a couple of examples. In Section 3 we discuss related work. In Section 4 we define our object language. The update calculus is introduced in Section 5, and a type system for the update calculus is developed in Section 6. Conclusions given in Section 7 complete this paper.

2 Update Programming

To give an impression of the concept of update programming we show some updates to Haskell programs and how they can be implemented in HULA, the Haskell Update LAnguage [8] that we are currently developing.

Suppose a programmer wants to extend a module for binary search trees by a `size` operation giving the number of nodes in a tree. Moreover, she wants to support this operation in constant time and therefore plans to extend the representation of the tree data type by an integer field for storing the information about the number of nodes contained in a tree. The definition of the original tree data type and an insert function are as follows:

```
data Tree = Leaf | Node Int Tree Tree

insert :: Int -> Tree -> Tree
insert x Leaf         = Node x Leaf Leaf
insert x (Node y l r) = if x<y then Node y (insert x l) r
                               else Node y l (insert x r)
```

The desired program extension requires a new function definition `size`, a changed type for the `Node` constructor (since a leaf always contains zero nodes, no change for this constructor is needed), and a corresponding change for all occurrences of `Node` in patterns and expressions. Adding the definition for the size function is straightforward and is not very exciting from the update programming point of view. The change of the `Node` constructor is more interesting since the change of its type in the `data` definition has to be accompanied by corresponding changes in all `Node` patterns and `Node` expressions. We can express this update as follows.

```
con Node : {Int} t in
    (case Node {s} -> Node {succ s}
        | Leaf -> Node {1}); Node {1}
```

The update can be read as follows: the `con` update operation adds the type `Int` as a new first parameter to the definition of the `Node` constructor. The notation $a\,\{r\}\,b$ is an abbreviation for the rewrite rule $a\,b \rightsquigarrow a\,r\,b$. So `{Int} t` means extend the type `t` on the left by `Int`. The keyword `in` introduces the updates that apply to the scope of the `Node` constructor. Here, a `case` update specifies how to change all pattern matching rules that use the `Node` constructor: `Node` patterns are extended by a new variable `s`, and to each application of the `Node` constructor in the return expression of that rule, the expression `succ s` is added as a new first argument (`succ` denotes the successor function on integers, which is predefined in Haskell). The `Leaf` pattern is left unchanged, and occurrences of the `Node` constructor within its return expression are extended by `1`. As an alternative to the `case` update, the rule `Node {1}` extends all other `Node` expressions by `1`.

The application of the update to the original program yields the new object program:

```
data Tree = Leaf | Node Int Int Tree Tree

insert :: Int -> Tree -> Tree
insert x Leaf          = Node 1 x Leaf Leaf
insert x (Node s y l r) =
       if x<y then Node (succ s) y (insert x l) r
              else Node (succ s) y l (insert x r)
```

It is striking that with the shown definition the `case` update is applied to all case expressions in the whole program. In our example, this works well since we have only one function definition in the program. In general, however, we want to be able to restrict `case` updates to specific functions or specify different `case` updates for different functions. This can be achieved by using a further update operation that performs updates on function definitions:

```
con Node : {Int} t in
    fun ‘insert x y:
        (case Node {s} -> Node {succ s}
            | Leaf -> Node {1}); Node {1}
```

This update applies the `case` update only to the definition of the function `insert`. Here the backquote is used to distinguish Haskell variables from meta variables of the update language.[2] Uses of the function `insert` need not be updated, which is indicated by the absence of the keyword `in` and a following update. We can add further `fun` updates for other functions in the program to be updated each with its own `case` update. Note that the variables `x` and `y` of the update language are meta variables with respect to Haskell that match any object (that is, Haskell) variable.

We can observe a general pattern in the shown program update: a constructor is extended by a type, all patterns are extended at the (corresponding position) by a new variable, and expressions built by the constructor are extended either by a function which is applied to the newly introduced variable (in the case that

[2] The backquote is not needed for `succ` and `s` since they appear as free variables in RHSs of rules, which means that they cannot reasonably be meta variables since they would be unbound. Therefore they are automatically identified as object variables.

the expression occurs in the scope of a pattern for this constructor) or by an expression. We can define such a generic update, say `extCon`, once and store it in an update library, so that constructor extensions as the one for `Node` can be expressed as applications of `extCon` [8]. For example, the size update can then be expressed by:

```
extCon Node Int succ 1
```

which would have exactly the effect as the update shown above. We plan to implement extensions to text editors like Emacs or Vim that offer generic type-correctness preserving updates like renaming or `extCon` via menus.

Of course, it is very difficult (if not generally impossible) to write generic update programs that guarantee overall semantic correctness. Any change to a program requires careful consideration by the programmer, and this responsibility is still required when using update programs. We do not claim to free the update process from any semantics consideration; however, we do claim that update programs make the update process more reliable by offering type-preservation guarantees and consistency in updates.

Other examples, such as generalizing function definitions or a collection of updates to maintain variations of a lambda-calculus implementation are discussed in [9] where we also indicate how update programming could be applied to Java.

3 Related Work

There is a large body of work on impact analysis that tries to address the problems that come with performing changes to software [2,4]. However, we know of no work that attempts to exploit impact analysis to perform fully automated software changes.

Performing structured program updates is supported by program editors that can guarantee syntactic or even type correctness and other properties of changed programs. Examples for such systems are Centaur [6], the synthesizer generator [11], or *CYNTHIA* [15]. 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.

We have introduced a language-based view of program updates in [7]. Viewing programs as abstract data types goes beyond the idea of syntax-directed program editors because it allows a programmer to combine basic updates into update programs that can be stored, reused, changed, shared, and so on. The update programming approach has, in particular, the following two advantages: First, we can work on program updates offline, that is, once we have started a program change, we can pause and resume our work at any time without affecting the object program. Although the same could be achieved by using a program editor together with a versioning tool, the update program has the advantage of much better reflecting the changes performed so far than a partially changed object program that only shows the result of having applied a number of update steps. Second, independent updates can be defined and applied independently. For example, assume an update u_1 followed by an update u_2 (that does not depend on or interfere with u_1) is applied to a program. With the editor approach, we can undo u_2 and also u_2 and u_1, but we cannot undo just u_1 because the changes performed by u_2 are only implicitly contained in the final version that has to be discarded to undo u_1. In contrast, we can undo each of the two updates with the proposed update programming approach by simply applying only the other update to the original program.

Programs that manipulate programs are also considered in the area of meta programming [12]. However, existing meta programming systems, such as MetaML [13], are mainly concerned with the generation of programs and do not offer means for analyzing programs (which is needed for program transformation). Refactoring [10] is an area of fast-growing interest. Refactoring (like the huge body of work on program optimization and partial evaluation) leaves the semantics of a program unchanged. Program transformations that change the behavior of programs are also considered in the area of aspect-oriented programming [1], which is concerned with performing "cross-cutting" changes to a program.

Our approach is based in part on applying update rules to specific parts of a program. There has been some work in the area of term rewriting to address this issue. The ELAN logical framework introduced a strategy language that allows users to specify their own tactics with operators and recursion [5]. Visser has extended the set of strategy operators and has put all these parts together into a system for program transformation, called *Stratego* [14]. These proposals allow a very flexible specification of rule application strategies, but they do not guarantee type correctness of the transformed programs.

A related approach that is concerned with type-safe program transformations is pursued 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 [3]. In particular, he describes a type system for dependent types for this language. However, in his system symbols must retain their types over transformations whereas in our approach it is possible that symbols change their type (and name).

4 The Object Language

To keep the following description short and simple, we use lambda calculus together with a standard Hindley/Milner type system as the working object language. The syntax of lambda-calculus expressions and types is shown in Figure 1. In addition to expressions e, types t, and type schemas s, we use c to range over constants, v to range over variables, and b over basic types. The definition of the type rules is standard and is omitted for lack of space.

Since the theory of program updates is independent of the particular dynamic semantics of the object language (call-by-value, call-by-need, ...), we do not have to consider a dynamic semantics.

$$\begin{aligned} e &::= c \mid v \mid e\ e \mid \lambda v.e \mid \texttt{let}\ v\,\texttt{=}\,e\ \texttt{in}\ e \\ t &::= b \mid a \mid t \rightarrow t \\ s &::= t \mid \forall \bar{a}.t \end{aligned}$$

Fig. 1. Syntax and types of lambda calculus.

The main idea to achieve a manageable update mechanism is to perform somehow "coordinated" updates of the *definition* and all corresponding *uses* of a symbol in a program. We therefore consider the available forms of symbol definitions more closely. In general, a definition has the following form:

$$\texttt{let}\ v\,\texttt{=}\,d\ \texttt{in}\ e$$

where v is the symbol (variable) being defined, d is the defining expression, and e is the scope of the definition, that is, e is an expression in which v will be used with the definition d (unless hidden by another nested definition for v). We call v the *symbol*, d the *defining expression*, and e the *scope* of the definition. If no confusion can arise, we sometimes refer to d also as the *definition* (of v). β-redexes also fit the shape of a definition since a (non-recursive) $\texttt{let}\ v\,\texttt{=}\,d\ \texttt{in}\ e$ is just an abbreviation for $(\lambda v.e)\ d$.

Several extensions of lambda calculus that make it a more realistic model for a language like Haskell also fit the general pattern of a definition, for example, data type/constructor definitions and pattern matching rules. We will comment on this in Section 5.2.

5 The Update Calculus

The update calculus basically consists of rewrite rules and a scope-aware update operation that is able to perform updates of the definition and uses of a symbol. In addition, we need operations for composing updates and for recursive application of updates.

5.1 Rules

A rewrite rule has the form $l \rightsquigarrow r$ where l and r are expressions that might contain meta variables (m), that is, variables that are different from object variables and can represent arbitrary expressions. Expressions that possibly contain meta variables are called *patterns*. The type system for lambda calculus has to be extended by a rule for meta variables that is almost identical to the rule for variables (except that meta variables have monomorphic types).

An update can be performed on an expression e by applying a rule $l \rightsquigarrow r$ to e which means to match l against e, which, if successful, results in a binding σ (called *substitution*) for the meta variables in l. The fact that a pattern like l matches an expression e (under the substitution σ) is also written as: $l \succ e$ ($l \underset{\sigma}{\succ} e$). We assume that l is linear, that is, l does not contain any meta variable twice.

The result of the update operation is $\sigma(r)$, that is, r with all meta variables being substituted according to σ. If l does not match e, the update described by the rule is not performed, and e remains unchanged.

We use the matching definitions and notations also for types. If a type t matches another type t' (that is, $t \succ t'$), then we also say that t' is an instance of t.

5.2 Update Combinators

We can build more complex updates from rules by alternation and recursion. For example, the *alternation* of two updates u_1 and u_2, written as $u_1 \,;\, u_2$, first tries to perform the update u_1. If u_1 can be applied, the resulting expression is also the result of $u_1 \,;\, u_2$. Only if u_1 does not apply, the update u_2 is tried. *Recursion* is needed to move updates arbitrarily deep into expressions. For example, since a rule is always tried at the root of an expression, an update like $\texttt{1} \rightsquigarrow \texttt{2}$ has no effect when applied to the expression `1+(1+1)`. We therefore introduce a recursion operator $\downarrow$ that causes its argument update to be applied (in a top-down manner) to all subexpressions. For example, the update $\downarrow(\texttt{1} \rightsquigarrow \texttt{2})$ applied to `1+(1+1)` results in the expression `2+(2+2)`. (We use the recursion operator only implicitly in scope updates and do not offer it to the user.)

In a *scope update*, each element of a definition $\texttt{let}\ v = d\ \texttt{in}\ e$, that is, v, d, or e, can be changed. Therefore, we need an update for each part. The update of the variable can just be a simple renaming, but the update of the definition and of the scope can be given by arbitrarily complex updates. We use the syntax $\{v \rightsquigarrow v' \colon u_d\}u_u$ for an update that renames v to v', changes v's definition by u_d, and all of its uses by u_u. (We also call u_d the *definition update* and u_u the *use update*.) Note that u_u is always applied recursively, whereas u_d is only applied to the root of the definition. However, to account for recursive `let` definitions we apply u_u also recursively to the result obtained by the update u_d. We use x to range over variables (v) and meta variables (m), which means that we can use a scope update to update specific bindings (by using an object variable) or to apply to arbitrary bindings (by using a meta variable). Either one of the variables (but not both) can be missing. These special cases describe the creation or removal of a binding. In both cases, we have an expression instead of a definition update. This expression is required in the case of binding removal where it is used to replace all occurrences of the removed variable. (Note that e must neither contain x nor a possible object variable that matches x in case x is a meta variable.) In the case of binding creation, e is optional and is used, if present, to create an expression $\texttt{let}\ v = e\ \texttt{in}\ e''$ where e'' is the result of applying u to e'. Otherwise, the result is $\lambda v.e''$. The syntax of updates is shown in Figure 2.

We use an abbreviated notation for scope updates that do not change names, that is, we write $\{v \colon u_d\}u_u$ instead of $\{v \rightsquigarrow v \colon u_d\}u_u$. The updates of either the defining expression or the scope can be empty, which means that there is no update for that part. The updates are then simply written as $\{v \colon u_d\}$ and $\{v\}u_u$, respectively, and are equivalent to updates $\{v \colon u_d\}\iota$ and $\{v \colon \iota\}u_u$, respectively.

$u ::= \iota$	Identity (No Update)
$\mid\ p \leadsto p$	Rule
$\mid\ \{x \leadsto x\colon u\}u$	Change Scope
$\mid\ \{\leadsto v[= e]\}u$	Insert Scope
$\mid\ \{x \leadsto e\}u$	Delete Scope
$\mid\ u\,;\,u$	Alternative
$\mid\ \downarrow u$	Recursion

Fig. 2. Syntax of updates.

Let us consider some examples. We already have seen examples for rules. A simple example for change scope is an update for consistently renaming variables $\{\mathtt{v} \leadsto \mathtt{w}\}\mathtt{v} \leadsto \mathtt{w}$. This update applies to a lambda- or let-bound variable `v` and renames it and all of its occurrences that are bound by that definition to `w`. The definition of `v` is usually not changed by this update. However, if `v` has a recursive definition, references to `v` in the definition will be changed to `w`, too, because the use update is also applied to the definition of a symbol.

A generalization of a function `f` can be expressed by the update $u = \{\mathtt{f}\!:\!\{\leadsto\mathtt{w}\}\mathtt{1}\leadsto\mathtt{w}\}\mathtt{f}\leadsto\mathtt{f\ 1}$. u is a change-scope update for `f`, which does not rename `f`, but whose definition update introduces a new binding for `w` and replaces all occurrences of a particular constant expression (here `1`) by `w` in the definition of `f`. u's use update makes sure that all uses of `f` are extended by supplying a new argument for the newly introduced parameter. Here we use the same expression that was generalized in `f`'s definition, which preserves the semantics of the program.

To express the size update example in the update calculus we have to extend the object language by constructors and `case` expressions and the update calculus by corresponding constructs, which is rather straightforward (in fact, we have already implemented it in our prototype). An interesting aspect is that each alternative of a `case` expression is a separate binding construct that introduces bindings for variables in the pattern. The scope of the variables is the corresponding right hand side of the `case` alternative. Since these variables do not have their own definitions, we can represent a `case` alternative by a lambda abstraction—just for the sake of performing an update. A `case` update can then be translated into an alternative of change-scope updates. For example, the translation of the size update yields:

```
{Node:t ⇝ Int->t}
    ({Node}({⇝s}Node⇝Node (succ s));
     {Leaf}Node⇝Node 1);
    Node⇝Node 1
```

The outermost change-scope update expresses that the definition of the `Node` constructor is extended by `Int`. The use update is an alternative whose second part expresses to extend all `Node` expressions by `1` to accommodate the type change of the constructor. The first alternative is itself an alternative of two change-scope updates. (Since the `;` operation is associative, the brackets are strictly not needed.) The first one applies to definitions of `Node` which (by way of translation) can only be found in lambda abstractions representing `case` alternatives. The new-scope update will add another lambda-binding for `s`, and the use update extends all `Node` expressions by the expression `succ s`. The other alternative applies to lambda abstractions representing `Leaf` patterns.

This last example demonstrates that the presented update calculus is not restricted to deal just with lambda abstractions or `let` bindings, but rather can serve as a general model for expressing changes to binding constructs of all kinds.

Due to space limitations we omit here the formal definition of the semantics that defines judgments of the form $[\![u]\!]_\rho(e) = e'$, see the extended version of this paper [9].

6 A Type System for Updates

The goal of the type system for the update calculus is to find all possible type changes that an update can cause to an *arbitrary* object program. We show that if these type changes "cover" each other appropriately, then the generated object program is guaranteed to be type correct.

6.1 Type Changes

Since updates denote changes of expressions that may involve a change of their types, the types of updates are described by *type changes*. A type change (δ) is essentially given by a pair of types ($t{\leadsto}t$), but it can also be an alternative of other type changes ($\delta|\delta$). For example, the type change of the update $\texttt{1} \leadsto \texttt{True}$ is $\texttt{Int} \leadsto \texttt{Bool}$, while the type change of $\texttt{1} \leadsto \texttt{True}\,;\,\texttt{odd} \leadsto \texttt{2}$ is $\texttt{Int} \leadsto \texttt{True}|\texttt{Int->Bool} \leadsto \texttt{Int}$.

Recursively applied updates might cause type changes in subexpressions that affect the type of the whole expression. Possible dependencies of an expression's type on that of its subexpressions are expressed using the two concepts of *type hooks* and *context types*. For example, the fact that the type of `odd 1` depends on the type of `1` is expressed by the hook $\texttt{Int}{\hookrightarrow}\texttt{Bool}$, the dependency on `odd` is $\texttt{Int->Bool}{\hookrightarrow}\texttt{Bool}$. The dependency on the whole expression is by definition empty (ϵ), and a dependency on any expression that is not a subexpressions is represented by a "constant hook" ${\hookrightarrow}\texttt{Bool}$.

The application of a type hook C to a type t yields a *context type* denoted by $C\langle t\rangle$ that exposes t as a possible type in a type derivation. The meaning of a context type is given by the following equations.

$$
\begin{aligned}
\epsilon\langle t\rangle &= t \\
{\hookrightarrow}t_2\langle t\rangle &= t_2 \\
t_1{\hookrightarrow}t_2\langle t\rangle &= \begin{cases} t_2 & \text{if } t \succ t_1 \\ \text{error} & \text{otherwise} \end{cases}
\end{aligned}
$$

$$\begin{aligned} \delta &::= \tau \leadsto \tau \quad \big| \quad \delta|\delta \\ \tau &::= b \mid a \mid \tau \rightarrow \tau \mid C\langle\tau\rangle \mid \tau_{|C} \\ C &::= \epsilon \mid \hookrightarrow t \mid t \hookrightarrow t \end{aligned}$$

Fig. 3. Type changes.

The rationale behind context types is to capture changes of types that possibly happen only in subexpressions and do not show up as a top-level type change. Context types are employed to describe the type changes for use updates in scope updates. For example, the type change of the update $u' = \texttt{1} \leadsto \texttt{w}$ is $\texttt{Int} \leadsto \texttt{a}$. However, when u' is used as a use update of a scope update $u = \{\leadsto\texttt{w}\}\texttt{1} \leadsto \texttt{w}$, it is performed recursively, so that the type change is described using a context type $C\langle\texttt{Int}\rangle \leadsto C\langle\texttt{a}\rangle$.

To describe the type change for u, the type for the newly introduced abstraction has to be taken into account. Here we observe that the type of w cannot be a in general, because w might be, through the recursive application of the rule, placed into an expression context that constrains w's type. For example, if we apply u to `odd 1`, we obtain λ`w.odd w` where w's type has to be `Int`. In general, the type of a variable is constrained to the type of the subexpression that it replaces. We can use a type hook that describes a dependency on a type of a subexpression e to express a constraint on a type variable that might replace e. Such a *constraint type* is written as $\texttt{a}_{|C}$. Its meaning is to restrict a type variable a by the type of a subexpression (represented by the left part of a type hook):

$$\begin{aligned} a_{|t_1 \hookrightarrow t_2} &= t_1 \\ t_{|C} &= t \end{aligned}$$

The type change for u is therefore given by $C\langle\texttt{Int}\rangle \leadsto \texttt{a}_{|C}\texttt{->}C\langle\texttt{a}_{|C}\rangle$.

To see how type hooks, context types, and constrained types work, consider the application of u to `1`, which yields λ`w.w`. The corresponding type change $\texttt{Int} \leadsto \texttt{a->a}$ is obtained using the type hook ϵ. However, applied to `odd 1`, u yields λ`w.odd w` with the type change $\texttt{Bool} \leadsto \texttt{Int->Bool}$, which is obtained from the type hook $\texttt{Int}\hookrightarrow\texttt{Bool}$. As another example consider the renaming update $u = \{\texttt{x} \leadsto \texttt{y}\}\texttt{x} \leadsto \texttt{y}$. For the update we obtain a type change $C\langle\texttt{a}_{|C}\rangle \leadsto C\langle\texttt{b}_{|C}\rangle$ (which is the same as $C\langle\texttt{a}_{|C}\rangle \leadsto C\langle\texttt{a}_{|C}\rangle$). The type hook C results for the same reason as in the previous example. Applying u to the expression λ`x.1` yields λ`y.1` with a type change $\texttt{a->Int} \leadsto \texttt{a->Int}$, which can be obtained by using the type hook $\hookrightarrow\texttt{a->Int}$. Similarly, u changes λ`x.odd x` to λ`y.odd y` with a type change $\texttt{Int->Bool} \leadsto \texttt{Int->Bool}$. This type change is u's type change specialized for the type hook $\texttt{Int}\hookrightarrow\texttt{Int->Bool}$.

The syntax of contexts and type changes is summarized in Figure 3. Since the inference rules generate, in general, context constraints for arbitrary type changes, we have to explain how contexts are propagated through type changes to types:

$$\begin{aligned} C\langle\tau \leadsto \tau'\rangle &:= C\langle\tau\rangle \leadsto C\langle\tau'\rangle \\ C\langle\delta|\delta'\rangle &:= C\langle\delta\rangle|C\langle\delta'\rangle \end{aligned}$$

Types and type changes can be *applicative instances* of one another. This relationship says that a type t is an applicative instance of a function type $t' \to t$, written as $t \precsim t' \to t$. The rationale for this definition is that two updates u and u' of different types $t_1 \leadsto t_2$ and $t'_1 \leadsto t'_2$, respectively, can be considered well typed in an alternative $u\,;\,u'$ if one type change is an applicative instance of the other, that is, if $t_1 \leadsto t_2 \precsim t'_1 \leadsto t'_2$ or $t'_1 \leadsto t'_2 \precsim t_1 \leadsto t_2$, because in that case one update is just more specific than the other. For example, in the update

$$\{\texttt{f}\colon \texttt{succ} \leadsto \texttt{plus}\}\texttt{f}\ \underline{\texttt{x}} \leadsto \texttt{f}\ \underline{\texttt{x}}\ \texttt{1}\,;\,\texttt{f} \leadsto \texttt{f}\ \texttt{1}$$

the first rule of the alternative $\texttt{f}\ \underline{\texttt{x}} \leadsto \texttt{f}\ \underline{\texttt{x}}\ \texttt{1}$ has the type change $\texttt{Int} \leadsto \texttt{Int}$ whereas the second rule $\texttt{f} \leadsto \texttt{f 1}$ has the type change $\texttt{Int->Int} \leadsto \texttt{Int->Int}$. Still both updates are compatible in the sense that the first rule applies to more specific occurrences of `f` than the second rule. This fact is reflected in the type change $\texttt{Int} \leadsto \texttt{Int}$ being an applicative instance of $\texttt{Int->Int} \leadsto \texttt{Int->Int}$. The applicative instance relationship extends in a homomorphic way to all kinds of type changes and contexts.

Finally, note that a type change $t \leadsto t'$ does not necessarily mean that an update $u : t \leadsto t'$ maps an expression e of type t to an expression of type t', because u might not apply to e and thus we might get $[\![u]\!](e) = e$ of type t. Thus, the information about an update causing some type change is always to be interpreted as "optional" or "contingent on the applicability of the update".

6.2 Type-Change Inference

The type changes that are caused by updates are described by judgments of the form $\Delta \rhd u :: \delta$ where Δ is a set of *type-change assumptions*, which can take one of three forms:

(1) $x{\leadsto}x' :: t{\leadsto}t'$ expresses that x of type t is changed to x' of type t'. The following constraint applies: if x' is a meta variable, then $x' = x$ and $t' = t$.
(2) $v :_r t$ expresses that v is a newly introduced (object) variable of type t.
(3) $x :_\ell t$ expresses that x is a (object or meta) variable of type t that is only bound in the expression to be changed.

Type-change assumptions can be extended by assumptions using the "comma" notation as in the type system.

The type-change system builds on the type system for the object language. In the typing rule for rules we make use of projection operations that project on the left and right part of a type-change assumption. These projections are defined as follows:

$$\begin{aligned}\Delta_\ell &:= \{x : t \mid x{\leadsto}x' :: t{\leadsto}t' \in \Delta\} \cup \{x : t \mid x :_\ell t \in \Delta\}\\ \Delta_r &:= \{x' : t' \mid x{\leadsto}x' :: t{\leadsto}t' \in \Delta\} \cup \{x' : t' \mid x' :_r t' \in \Delta\}\end{aligned}$$

The type-change rules are defined in Figure 4. The rules for creating or deleting a binding have to insert a function argument type on either the right or the left

$$\rightsquigarrow_{\triangleright}\quad \frac{\Delta_\ell \vdash p : t \qquad \Delta_r \vdash p' : t'}{\Delta \triangleright p \rightsquigarrow p' :: t \rightsquigarrow t'} \qquad\qquad \iota_{\triangleright}\quad \frac{}{\Delta \triangleright \iota :: t \rightsquigarrow t}$$

$$;_{\triangleright}\quad \frac{\Delta \triangleright u :: \delta \quad \Delta \triangleright u' :: \delta' \quad \delta \vec{\precsim} \delta'' \quad \delta' \vec{\precsim} \delta''}{\Delta \triangleright u\,;\, u' :: \delta''} \qquad \frac{\Delta \triangleright u :: \delta \qquad \Delta \triangleright u' :: \delta'}{\Delta \triangleright u\,;\, u' :: \delta|\delta'}$$

$$\{:\}^{chg}_{\triangleright}\quad \frac{\Delta[, x\rightsquigarrow x' :: t_{|C}\rightsquigarrow t'_{|C}] \triangleright u_d :: t_{[|C]} \rightsquigarrow t'_{[|C]} \quad \Delta[, x\rightsquigarrow x' :: t_{|C}\rightsquigarrow t'_{|C}] \triangleright u_u :: \delta}{\Delta \triangleright \{x\rightsquigarrow x' : u_d\}u_u :: C\langle\delta\rangle}$$

$$\{:\}^{ins}_{\triangleright}\quad \frac{\{\bar{a}\} = FV(t) - FV(\Delta_r) \quad \Delta[, w : t_{[|C]}] \vdash e : \forall\bar{a}.t_{[|C]} \quad \Delta[, w :_r t_{|C}] \triangleright u :: \delta}{\Delta \triangleright \{\rightsquigarrow w = e\}u :: t_{[|C]} \underset{r}{\rightarrow} C\langle\delta\rangle}$$

$$\frac{\Delta[, w :_r t_{|C}] \triangleright u :: \delta}{\Delta \triangleright \{\rightsquigarrow w\}u :: t_{[|C]} \underset{r}{\rightarrow} C\langle\delta\rangle} \qquad \{:\}^{del}_{\triangleright}\quad \frac{\Delta[, x :_\ell t_{|C}] \triangleright u :: \delta \quad \Delta_r \vdash e : t_{[|C]}}{\Delta \triangleright \{x\rightsquigarrow e\}u :: t_{[|C]} \underset{\ell}{\rightarrow} C\langle\delta\rangle}$$

Fig. 4. Type change system.

part of a type change. This type insertion works across alternative type changes; we use the notation $\tau \underset{\ell}{\rightarrow} \delta$ ($\tau \underset{r}{\rightarrow} \delta$) to extend the argument (result) type of a type change to a function type. The definition is as follows.

$$\begin{aligned} \tau \underset{\ell}{\rightarrow} (\tau_l \rightsquigarrow \tau_r) &:= (\tau \rightarrow \tau_l) \rightsquigarrow \tau_r & \qquad \tau \underset{\ell}{\rightarrow} (\delta|\delta') &:= (\tau \underset{\ell}{\rightarrow} \delta)|(\tau \underset{\ell}{\rightarrow} \delta') \\ \tau \underset{r}{\rightarrow} (\tau_l \rightsquigarrow \tau_r) &:= \tau_l \rightsquigarrow (\tau \rightarrow \tau_r) & \qquad \tau \underset{r}{\rightarrow} (\delta|\delta') &:= (\tau \underset{r}{\rightarrow} \delta)|(\tau \underset{r}{\rightarrow} \delta') \end{aligned}$$

The inference rule $\rightsquigarrow_{\triangleright}$ connects the type system of the underlying object language (lambda calculus) with the type-change system.

We have several rules for scope updates. To save space we combine two rules for each case by using square brackets for optional rule parts. For example, in the rule $\{:\}^{ins}_{\triangleright}$ if and only if the premise can be proved without using the assumption for w, then there is no type hook C on the type t in the conclusion.

6.3 Soundness of the Update Type System

In this section we define a class of *well-structured updates* that will preserve the well-typing of transformed object-language expressions. An update that, when applied to a well-typed expression, yields again a well-typed expression is called *safe*. In other words, we will show that typeable well-structured updates are safe. The structure condition captures the following two requirements:

(A) An update of the definition of a symbol that causes a change of its type or its name is accompanied by an update for all the uses of that symbol (with a matching type change).

(B) No use update can introduce a non-generalizing type change, that is, for each use update that has a type change $t\rightsquigarrow t'|\delta$ we require that t is a generic instance of t' or that one type, t or t', is an applicative instance of the other.

Condition (A) prevents ill-typed applications of changed symbols as well as unbound variables whereas (B) prevents type changes from breaking the well typing of their contexts. An intuitive explanation of why these conditions imply safety for well-typed updates can be obtained by looking at all the possible ways in which an update can break the type correctness of an expression and how these possibilities are prevented by the type system or the well-structuring constraints. For a detailed discussion, see [9].

Let us now define the well-structuring constraint formally. We first identify some properties of change-scope updates. Let $u = \{x \rightsquigarrow x' \colon u_d\} u_u$ and let $x \rightsquigarrow x' :: t \rightsquigarrow t'$ be the assumption that has been used in rule $\{:\}^{chg}_{\triangleright}$ to derive its type change, say $t_1 \rightsquigarrow t_2 | \delta$.

(1) u is *self-contained* iff $x \neq x' \vee t \neq t' \implies \exists u, u', p : u_u = u\,;\, x \rightsquigarrow p\,;\, u'$.
(2) u is *smooth* iff $t' \succ t$ or $t \overset{\rightarrow}{\prec} t'$ or $t' \overset{\rightarrow}{\prec} t$
(3) u is (at most) *generalizing* iff $t_2 \succ t_1$

An update u is *well structured* iff it is well typed and all of its contained change-scope updates are self-contained, smooth, and generalizing.

When we consider the application of a well-structured update u to a well-typed expression e, the following two cases can occur: (1) u does not apply to e. In this case e is not changed by u and remains well typed. (2) u applies to e and changes it into e'. In this case we have to show that from the result type of u we can infer the type of e'. We collect the results in the following two lemmas.

Lemma 1. *u does not apply to $e \wedge \Gamma \vdash e : t \implies \Gamma \vdash [\![u]\!](e) : t$*

Lemma 2 (Soundness). *If u is well structured and applies to e, then*

$$\Delta \triangleright u :: \tau \rightsquigarrow \tau' | \delta \wedge \Delta_\ell \vdash e : \tau \implies \Delta_r \vdash [\![u]\!](e) : \tau'$$

The lemma expresses that the derivation of a type change that includes an alternative $\tau \rightsquigarrow \tau'$ ensures for any expression e of type τ that u transforms e into an expression of type τ'. We have to use τ in the lemma because the type change for u is generally given by context types. For a concrete expression e, the type inference will fix any type hooks, which allows τ to be simplified to a type t. Finally, we can combine both lemmas in the following theorem.

Theorem 1. *If u is well structured, then*

$$\Delta \triangleright u :: \tau \rightsquigarrow \tau' | \delta \wedge \Delta_\ell \vdash e : \tau \implies \Delta_r \vdash [\![u]\!](e) : t \wedge (t = \tau \vee t = \tau')$$

Let us consider the safety of some of the presented example updates. The function generalization update from Section 5.2 is safe, which can be checked by applying the definitions of "well structured" and the rules of the type-change system. The first size update (Section 2) is also safe, although to prove it we need the extension of lambda calculus by constructors and `case` expressions. In contrast, the second size update is *not* safe since the `case` update will be applied only to the definition of `insert` (and not to other functions).

7 Conclusions and Future Work

We have introduced an update calculus together with a type-change system that can guarantee the safety of well-structured updates, that is, well-typed, safe updates preserve the well typing of lambda-calculus expressions. The presented calculus can serve as the basis for type-safe update languages. Currently, we are working on the design and implementation of an update language for Haskell.

One area of future work is to relax the rather strict well-structuring conditions and facilitate larger classes of update programs under the concept of *conditional safety*, which means to infer constraints for object programs that are required for their type preservation under the considered update.

References

1. ACM. *Communications of the ACM*, volume 44(10), October 2001.
2. R. S. Arnold and S. A. Bohner. Impact Analysis – Towards a Framework for Comparison. In *IEEE Int. Conf. on Software Maintenance*, pages 292–301, 1993.
3. N. Bjørner. Type Checking Meta Programs. In *Workshop on Logical Frameworks and Meta-Languages*, 1999.
4. S. A. Bohner and R. S. Arnold, editors. *Software Change Impact Analysis.* IEEE Computer Society Press, Los Alamitos, CA, 1996.
5. B. Borovanský, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN: A Logical Framework Based on Computational Systems. In *Workshop on Rewriting Logic and Applications*, 1996.
6. 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.
7. M. Erwig. Programs are Abstract Data Types. In *16th IEEE Int. Conf. on Automated Software Engineering*, pages 400–403, 2001.
8. M. Erwig and D. Ren. A Rule-Based Language for Programming Software Updates. In *3rd ACM SIGPLAN Workshop on Rule-Based Programming*, pages 67–77, 2002.
9. M. Erwig and D. Ren. An Update Calculus for Type-Safe Program Changes. Technical Report TR02-60-09, Department of Computer Science, Oregon State University, 2002.
10. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Reading, MA, 1999.
11. T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1989.
12. T. Sheard. Accomplishments and Research Challenges in Meta-Programming. In *2nd Int. Workshop on Semantics, Applications, and Implementation of Program Generation*, LNCS 2196, pages 2–44, 2001.
13. W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.
14. E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *12th Int. Conf. on Rewriting Techniques and Applications*, 2001.
15. J. Whittle, A. Bundy, R. Boulton, and H. Lowe. An ML Editor Based on Proof-as-Programs. In *9th PLILP*, LNCS 1292, pages 389–405, 1997.