Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 67 (2007) 199–222
Science of Computer Programming
www.elsevier.com/locate/scico

An update calculus for expressing type-safe program updates

Martin Erwig*, Deling Ren

Oregon State University, School of EECS, Corvallis, OR 97331, USA

Received 8 December 2004; received in revised form 14 February 2006; accepted 27 January 2007
Available online 1 April 2007

Abstract

The dominant share of software development costs is spent on software maintenance, particularly the process of updating programs in response to changing requirements. Currently, such program changes tend to be performed using text editors, an unreliable method that often causes many errors. In addition to syntax and type errors, logical errors can be easily introduced since text editors cannot guarantee that changes are performed consistently over the whole program. All these errors can cause a correct and perfectly running program to become instantly unusable. It is not surprising that this situation exists because the "text-editor method" reveals a low-level view of programs that fails to reflect the structure of programs.

We address this problem by pursuing a programming-language-based approach to program updates. To this end we discuss in this paper the design and requirements of an update language for expressing update programs. We identify as the essential part of any update language a *scope update* that performs coordinated update of the definition and all uses of a symbol. As the underlying basis for update languages, we define an update calculus for updating lambda calculus programs. We develop a type system for the update calculus 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. The update calculus can serve as a basis for higher-level update languages, such as for Haskell or Java.

Keywords: Program transformation; Meta programming; Type safety; Refactoring; Software evolution

1. Introduction

1.1. Current problems with software maintenance

Most software products are dynamic entities that undergo many changes through their lifetime, or as Lehman puts it [27]: "There is no such thing as a 'finished' computer program". The cost of maintaining software dominates the overall cost of software: it is estimated that maintenance requires more than 60% of all software development effort [34]. Some estimates give figures up to 70% [2,42] or even 80% [44]. It has also been reported that the cost of software maintenance is growing [16] and that maintenance costs grow at a rate of 10% a year [5]. Changes to software that are performed in the course of software maintenance have been classified into corrective, preventive, adaptive, and

* Corresponding author. Tel.: +1 541 737 8893; fax: +1 541 737 3014.
E-mail address: erwig@eecs.oregonstate.edu (M. Erwig).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2007.01.003

perfective [28,42]. The latter two categories comprise changes in response to a changing environment and to new requirements. Together they account for 75% to 80% of all maintenance cost [28,34]. Significantly improving the process of changing software can greatly reduce the overall cost of software maintenance and development, as well as substantially increasing product reliability and efficiency.

Changes to software programs today are usually made using text editors, a process that often causes problems. For example, syntax and type errors are frequently introduced when a few minor changes are made to a correct program. Also serious is the introduction of *logical* errors, which cannot be detected by a compiler. In this instance, if a change to a certain expression (that does not change its type) is not performed consistently, the program might still typecheck, but part of its computation is erroneous. Even if the required program changes are minimal, inconsistencies can be introduced quite easily through editing operations, and worse, go unnoticed. The fatal aspect of this common situation is that a correct and perfectly running program can become unusable in a few seconds.

This unfortunate circumstance is not surprising because performing program changes with a text editor reveals a very low-level program view, namely that of character sequences. Moreover, the offered operations on programs are basically those of inserting and deleting characters in the program's textual representation. This view does not reflect the structure of programs, and it follows that adding or deleting single characters are not the right operations on programs. The lack of tools for automating software changes has been identified as a key problem by several researchers [44,42,38]. Takang and Grubb write [42], "The task of software maintenance is such a vital and complex one that it can no longer be done effectively without automated support".

1.2. Improving the software update process

We can view a software program as an element of an abstract data type (ADT) [12]. Then changes to software programs can be performed by applying ADT operations. Basic update operations can be combined through update combinators to build 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' 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-preserving 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.

Viewing programs as abstract data types also 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. As will be demonstrated in Section 2, we could actually use program updates as a basis to create a new kind of syntax-aware versioning tool that can inform much better about program changes than character-based programs like `diff`. 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.

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 [15] or by all kinds of so-called "cross-cutting" concerns in the fast-growing area of aspect-oriented programming [1,19,11,31,4], 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.

Although we propose update programming as a new approach to performing software changes, our goal is *not* to completely replace the use of text editors for programming; rather, we would like to complement them since there are many small, simple changes that can probably be accomplished most easily 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. The often described phenomenon of "resistance to change" makes this situation even less likely [3,10]. 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 metaprogramming, 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 experts and used by a much wider audience of programmers (for example, through 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.

1.3. The structure of this paper

In the next section we illustrate the idea of update programming with a couple of examples. In Section 3 we discuss related work. The notion of safety with regard to update programming is explained in Section 4. As a concrete example of a safe update programming language we will describe a calculus for updating lambda expressions in Section 5. In Section 5.1 we define the object language. The update calculus is introduced in Section 5.2, and a type system for the update calculus is developed in Section 5.3. We show the safety of the update calculus in Section 5.3.3. After outlining several directions for future research in Section 6, we will present some conclusions in Section 7.

2. Examples of update programming

We illustrate the idea of update programming through several examples. To this end we describe how updates to Haskell programs can be implemented in HULA, the Haskell Update LAnguage [13] that we are currently developing.

Suppose we want to extend a module for binary search trees by a `size` operation giving the number of nodes in a tree. Moreover, we want to support this operation in constant time, therefore we 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 desired program extension goes beyond the idea of refactoring, which is concerned with semantics-preserving restructurings of programs, and illustrates that update programming has applications that are more general than refactorings. The definition of the original tree data type and an insert function are as follows.

```
data Tree a = Leaf | Node a Tree Tree

insert :: Ord a => a -> Tree a -> Tree a
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 program update 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

[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.

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 where
    (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 **where** 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 object 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 all 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 following object program.

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

insert :: Ord a => a -> Tree a -> Tree a
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 behavior can be achieved by using a further update operation that performs updates on function definitions:

```
con Node : {Int} t where
    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.

Uses of the function insert need not be updated, which is indicated by the absence of the keyword **where** 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 metavariables 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 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* [13]. For example, the size update can then be expressed by the following expression, which would have exactly the effect as the update shown above.

```
extCon Node Int succ 1
```

One can imagine extensions to text editors like Emacs or Vim that offer generic type correctness preserving updates like renaming or *extCon* via menus. The update could then be performed as follows. The programmer selects the item "Extend Constructor" from a menu of program updates. This constructor update requires the name of the constructor

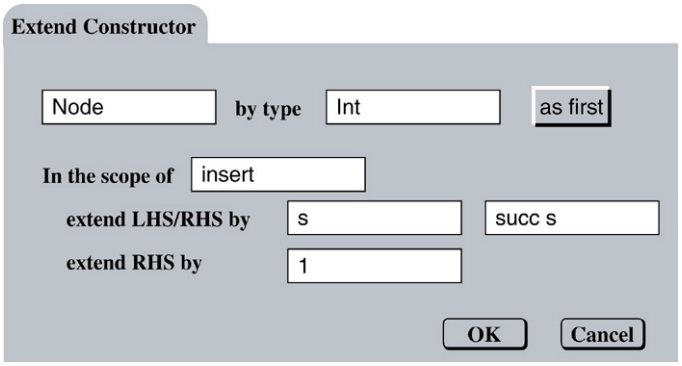

Fig. 1. Interactive update specification.

to be updated, the type by which it is to be extended, and a specification of how all the occurrences of the constructor in the program – used either in a pattern or in an expression – are to be extended to match the changed type. As a result of selecting the update from the menu, a window pops up that asks for further details about the update (see Fig. 1).

If the cursor is positioned on a constructor at the time the menu is invoked, the system assumes that this constructor should be changed and inserts the constructor name (here, `Node`) automatically. Moreover, the system can search for all places in the program where the constructor is used and provides a list of scopes that require a change—in the case of a constructor update these scopes are (possibly nested) function names. In our example there is just one scope, namely the function `insert`, but in typical binary tree module we would get a longer list of scopes, including functions, such as `delete` or `find`.[2] For each scope we now have to specify how pattern/expression pairs or just expressions are to be updated. The combined specification of a pattern/expression pair is needed since we might introduce a variable into the pattern that is to be used in the corresponding result expression. For example, in the `insert` function we want to extend the `Node` pattern by a variable `s` for the size, and we want to use `s` in the return expression for that pattern to extend the `Node` constructor by an expression `succ s` to express that the size of the tree is increased by one after the insertion of an element. On the other hand, we also have to be able to express the extension of constructors that occur in return expressions that do not have a `Node` constructor in the pattern of their LHS. In our example we want to set the size of any newly constructed tree to be one.

Although the just described update might look simple, some non-trivial computations take place behind the scenes—for example, the types of the expressions entered into the LHS/RHS boxes must match the type by which the constructor is extended, or any new variables that are introduced must not conflict with already existing variables. If we had used `x` instead of `s`, the system could rename it into `x'` (also the use in the expression would then be renamed to `succ x'`), or not apply the update at all and report an error, depending on the user's preferences regarding automatic renaming.

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 still exists 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.

The idea of update programming is, of course, not limited to Haskell. To illustrate how the described concepts can be employed in other programming languages, we describe a Java version of the presented tree example. In Java, the abstract data type can be defined as a class `Tree`, with a constructor `Tree` and an `insert` method. We omit the implementation details of `insert`, because they are not needed to illustrate the idea of update programming.

```
public class Tree
{
    Object content;
```

[2] Scopes can be combined so that an update can be specified once and can still be applied at different places. However, scopes cannot be deleted since a safe update must care for *all* occurrences of the constructor. Finally, if there is only one scope, this can then as well be omitted; the update is then like a global update.

```
    Tree left, right;

    public Tree ()
    {
        content = null;
        left    = null;
        right   = null;
    }
    public void insert (Tree t)
    {
        ...
        return;
    }
}
```

To extend the class with a tag for size, we add a field `size` and make corresponding changes to affected member methods. In this case, the class constructor `Tree` needs to be changed so that `size` is initialized when the tree is constructed; the `insert` method needs to be changed so that `size` is increased whenever a node is added. An update program could be defined as follows:

class `Tree :` {`int size;`} *d* **where**
    meth `insert :` {`size ++;`} *d*
    cons `Tree :` {`size = 0;`} *d*

In all three cases the metavariable *d* matches the definition of the class, method, and constructor, respectively. The expressed updates are simply inserting a variable declaration (in the case of the class update) and a variable assignment (in the case of the method and constructor update) in front of the existing definitions. Applying the update to the original program will yield the following updated Java program.

```
public class Tree
{
    int size;
    Object content;
    Tree left, right;

    public Tree ()
    {
        size = 0;
        content = null;
        left    = null;
        right   = null;
    }
    public void insert (Tree t)
    {
        size ++;
        ...
        return;
    }
}
```

To give another example, consider the task of generalizing a function definition, which works by identifying expressions in a function definition that should be made variable. We can express this update by an update function *genFun* that takes two parameters, the name of the function to be generalized (*f*) and the expression *e* to be generalized within *f*'s definition. The generalization works essentially in three steps. First, a new parameter `x` is added to the

definition of *f*, as the *i*th parameter, which is expressed by placing {*i*|x} after the function name *f*. Second, the expression *e* to be abstracted is replaced everywhere in *f*'s definition by the new variable x. The replacement is expressed using the substitution notation {*e*/x} (read: "replace *e* by x"). Two special cases of this substitution notation are {*e*/} ("delete *e*") and {x} ("insert x"), which we have already seen in the size update. Finally, all applications of the function are to be extended by adding a new argument, which we choose to be the abstracted expression so that the meaning of the original program is preserved. In HULA this generic update is defined by:

```
genFun f i e = fun f {i|x} : {e/x} where f {e}
```

Another application for update programming is the maintenance of programs that have many variants. For example, there exist many different forms of lambda calculus. We can use the following update to extend a data type `Lam` for representing lambda expressions (containing constructors only for variables, application, and lambda abstraction) and a corresponding evaluation function `eval` by constants:

```
data Lam : cs {Con String} where
     fun eval : case x of rs {Con c -> Con c}
     fun subst : case x of rs {Con c -> Con c}
```

All other functions that do a pattern matching on the data type `Lam` also need to be updated, such as the `subst` function here. The expression {`Con String`} expresses to add a new constructor named `Con` of argument type `String` to the data type definition of `Lam`. The new constructor definition is added after the existing constructors, which is expressed by placing the expression after the metavariable *cs* that matches the existing constructors. The update of the uses of the (changed) data type definition specifies only to update the function definition of `eval`. This happens by inserting a case rule after all other existing rules, which are bound to *rs*. The added rule expresses that constants evaluate to themselves.

With a similar update we can extend the lambda calculus implementation by `let` expressions:

```
data Lam : cs {Let String Lam Lam} where
     fun eval : case x of rs {Let v d e -> subst d v e}
     fun subst : case x of rs {Let v d e -> ... }
```

The definition of function `subst` is not relevant and therefore omitted here for simplicity. We can apply both updates independently or one after the other (in any order) to obtain a version of lambda calculus with constants *and* `let` expressions. If the original lambda calculus implementation changes, we can reapply the update programs to propagate the changes through the defined extensions. We can perform similar updates for extending type inference or other functions as well.

3. Related work

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 [9,37], the synthesizer generator [36], or *CYNTHIA* [51,50]. 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. An interesting observation is that the approach taken in the ML editor *CYNTHIA* is more powerful than other approaches since it is based on a richer representation of programs, that is, it exploits the Curry–Howard isomorphism [18,30], which directly relates proofs of type correctness with programs. In this respect it is very similar to proof editors like ALF [29], however, in contrast to ALF, proofs are not the main objective of *CYNTHIA* but rather used as a glueing representation between programs and their properties.

In [12] we have introduced a language-based view of program updates. One part of that work is the development of a general model of programs, updates, and the preservation of arbitrary properties. We have also discussed a way of ensuring type correctness for the simply typed lambda calculus that is based on computing required and provided changes in type assumptions. In [14] we have introduced an update calculus for the implicitly typed lambda calculus together with a type-change system that can infer possible type changes caused by updates. An extended version of this system is described in Section 5. In [13] we have defined an update language for a subset of Haskell. We also have shown how to translate this update language into an update calculus.

As Klint et al. point out in [21], evolutionary transformation of software have not yet received much attention except for refactoring. Our paper addresses program transformations beyond refactoring, in particular, those that allow type changes and semantics changes.

Programs that manipulate programs are also considered in the area of metaprogramming [39]. However, existing metaprogramming systems, such as MetaML [41] or Template Haskell [40], are mainly concerned with the generation of programs and do not offer means for analyzing programs (which is needed for program transformation). In fact, in a recent overview only a few source-level program transformations have been reported [49]. Among these, only software rephrasing and refactoring work on one and the same language. Refactoring [15] is an area of fast-growing interest with a few existing tools to perform refactoring automatically [38]. 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. AspectJ [19] and, Hyper/J [31] are two of the existing tools that can be used to deal with aspects in Java programs. These tools are used to merge a cross-cutting concern into one particular object program at a time. It is not possible, for example, to compile and typecheck aspects independently of programs to obtain type-safe reusable transformations. Composition filters [4] have been defined as a general extension for the object-oriented programming model.

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. Traditionally, rewrite systems consider the strategy in which rewrite rules are applied to be more or less fixed. In theorem proving *tactics* have been introduced to overcome the limitations of having only fixed strategies [32]. The ELAN logical framework introduced in addition to a fixed set of tactics a strategy language that allows users to specify their own tactics with operators and recursion [7,8]. Visser has extended the set of strategy operators by generic term traversals [48], pattern matching operators [45], and other rewrite strategies that are specifically useful for language processing [46] and has put all these parts together into a system for program transformation, called *Stratego* [47]. These proposals allow a very flexible specification of rule application strategies, but they do not guarantee syntax or type correctness of the transformed programs. Strafunski [26,25] and the so-called "Scrap Your Boilerplate" approach [23,24] are program-transformation systems that are based on generic traversal operations that can be extended/modified through combinators to create customized transformations. In particular, these combinators allow a programmer to specify transformations at specific parts of a program while not having to be concerned about the remaining syntactic structure, which is taken care of automatically through the default behavior of the traversal operation. Since object languages are represented as Haskell data types, updates written in these two approaches do preserve syntactic correctness of object programs, but they cannot guarantee the type correctness of the generated programs. In [43] Klint et al. extend term rewriting with traversal functions to traverse a tree automatically, according to a set of built-in traversal primitives. Their approach eliminates the need for extra rules for carrying data around and having non-sort-preserving transformations.

A specific task related to updates on programs is *program integration*, which is concerned with the combination of two variants A and B of a program P. The algorithm developed by Horwitz and others detects whether the updates that lead from P to A and B, interfere and if not, combines these updates into a single program that includes all functionality from P as well as the changes from A and B [17]. Algebraic properties of such a program integration operation have been studied by Reps [35].

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 [6]. 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 and essential that symbols can change their types (and names).

Lämmel describes a transformation-based approach for evolution of rule-based programs [22]. He introduces a suite of operators for the transformation. In his paper, evolution relations are defined as a measurement between the original program and evolved program. Certain properties are studied and identified for the transformation operators. One particularly interesting relation is type preservation in the sense of type equality of the predicates and functors between the original program and evolved program. However, this type preservation refers to that all symbols do not change their types, therefore type correctness is ensured. In this paper, we go one step further to allow variables change their types while the new program remains type correct.

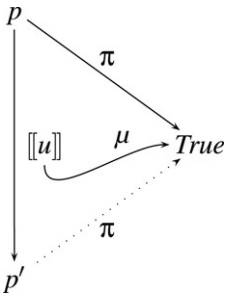

Fig. 2. Safe update theorems.

In [20] Klint describes a meta-environment based on the formalism ASF+SDF. A meta-environment is an interactive development environment for formal language definitions and for generating and testing particular programming environments. This meta-environment is a potential candidate for the implementation of the update calculus.

4. Safety of program updates

In this section we describe a generic language-based view of program updates together with a notion of safety for program updates.

A program p to be updated is an element of a language P, also called *object language*. A property π on a language P is given by a boolean function on P, that is, properties are propositions about programs. The fact that p is correct with respect to the property π is written as $\pi(p)$. In that case we say that p is π*-correct* or π*-valid*. Updates are specified in an update language U. The meaning of an update program u is a function $[\![u]\!]$ on object programs, that is, $[\![\_]\!] : U \to P \to P$. The following definition introduces an essential concept of the update programming approach.

Definition 1 (*Safety of Updates*). An update u is *safe* with respect to the language property π (or, u is π-safe, for short) if and only if $\forall p \in P.\pi(p) \Longrightarrow \pi([\![u]\!](p))$.

Since U is a language, we can identify properties for U (as we did for P). We use μ to denote a property on U. To design an update language we have to find characterizations of safe updates, that is, we want to find properties μ such that μ-correctness implies π-safety. In other words, given P, π, and U, find μ such that $\mu(u) \Longrightarrow u$ is π-safe, or, expanding the definition of π-safety, $\forall u.\mu(u) \Longrightarrow (\forall p.\pi(p) \Longrightarrow \pi([\![u]\!](p)))$. This generic schema for theorems is illustrated in Fig. 2.

We can now investigate a variety of update languages for different object languages preserving different properties under varying criteria. We call any instance (P, π, U, μ) of this general setting an *update scenario*. The existence of safe update theorems serves as the key question for the significance of any update scenario because if μ-correctness implies π-safety, we can work on updates completely independently from the object programs to which they will be eventually applied. This property is as important as static typing is for object programs—the partial correctness of an update (in the sense of not producing an object program violating π) can be checked without having to execute it (that is, apply it to an object program).

The focus on type correctness as one important property that an update language should preserve is justified by empirical studies [33]. However, from a practical point of view, there are also other properties whose preservation is worthwhile. For example, the information about to what degree updates can preserve layout or comments of updated programs is an important criterion for offering these updates in a program editor because programmers would probably not like nor expect their programs to be completely reformatted.

In the next section we will investigate in some detail one instance of the described scenario with P = lambda calculus, π = type correctness, U = update calculus, and μ = type-change correctness + structural constraint.

5. An update calculus for lambda calculus

5.1. The object language

We consider lambda calculus together with a standard Hindley/Milner type system as the working object language. The syntax of lambda calculus expressions is shown in Fig. 3. In addition to expressions e, we use v to range over variables. For simplicity, we omit constants here.

$$e \quad ::= \quad v \mid e\ e \mid \lambda v.e \mid \texttt{let}\ v = e\ \texttt{in}\ e$$

Fig. 3. Abstract syntax of lambda calculus.

Types are built from type variables (denoted by a) and function types (see Fig. 4). Type schemas are used to enable polymorphic typing for `let`-bound variables. We abbreviate a list of type variables $a_1 \ldots a_n$ by $\bar{a}$.

$$\begin{array}{rcl} t & ::= & a \mid t \to t \\ s & ::= & t \mid \forall \bar{a}.t \end{array}$$

Fig. 4. Types for lambda calculus.

FV gives the set of free variables of an expression, a type, or a type environment. Likewise, BV computes bound variables. We denote by $[w/v]e$ the capture-free substitution of the variable v by the variable w in the expression e.

The type system defines judgments of the form $\Gamma \vdash e : t$ where Γ is a type assumption, that is, a mapping from variables (v) to type schemas (s). The inference rules shown in Fig. 5 are standard except for the rule $\text{META}_{\vdash}$, which defines the typing of metavariables and which is explained below in Section 5.2.1.

$$\text{VAR}_{\vdash}\ \frac{\Gamma(v) = \forall \bar{a}.t' \qquad [t_i/a_i]t' = t}{\Gamma \vdash v : t} \qquad \text{ABS}_{\vdash}\ \frac{\Gamma, v : t' \vdash e : t}{\Gamma \vdash \lambda v.e : t' \to t} \qquad \text{APP}_{\vdash}\ \frac{\Gamma \vdash e : t' \to t \qquad \Gamma \vdash e' : t'}{\Gamma \vdash e\ e' : t}$$

$$\text{LET}_{\vdash}\ \frac{\{\bar{a}\} = FV(t') - FV(\Gamma) \qquad \Gamma, v : t' \vdash e' : t' \qquad \Gamma, v : \forall \bar{a}.t' \vdash e : t}{\Gamma \vdash \texttt{let}\ v = e'\ \texttt{in}\ e : t} \qquad \text{META}_{\vdash}\ \frac{\Gamma(m) = t}{\Gamma \vdash m : t}$$

Fig. 5. Type system for lambda calculus.

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 in the context of this paper.

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 in more detail. In general, a definition has the following form:

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

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) $\texttt{let}\ v = d\ \texttt{in}\ e$ is just an abbreviation for $(\lambda v.e)\ d$. However, the treatment of `let` differs from functions in the type system (allowing polymorphism) and also in the update language since it allows recursive definitions.

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 have demonstrated this idea by examples in Section 2, and we will explain the relationship in more detail based on the presented update calculus in Section 5.2.2.

5.2. 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 alternative updates and for recursive application of updates.

5.2.1. Rules

A rewrite rule has the form:

$$l \rightsquigarrow r$$

where l and r are *patterns*, which are basically expressions that might contain *metavariables* (m). Metavariables are different from object variables and can represent arbitrary expressions. The syntax of patterns is defined in Fig. 6.

$$p \quad ::= \quad m \mid v \mid p\ p'$$

Fig. 6. Patterns.

If we remove metavariables, patterns reduce to expressions that do not introduce bindings. Binding constructs will be updated by a special form, *scope update*, that takes care of the peculiarities of free/bound/fresh variables and their types that can occur with updates. Therefore, rewrite rules are restricted to patterns. The typing rule for metavariables in Fig. 5 is similar to the rule VAR$_{\vdash}$, but we only allow the binding of types (and not type schemas).

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 metavariables in l. Formally, a substitution is a mapping from variables to expressions. The fact that a pattern like l matches an expression e (under the substitution σ) is also written as: $l \succ e$ ($l \succ_{\sigma} e$). We assume that l is linear, that is, l does not contain any metavariable twice. The result of the update operation is $\sigma(r)$, that is, r with all metavariables 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.

5.2.2. Update combinators

More complex updates can be built from rules by alternation and recursion. For example, the *alternation* of two updates u_1 and u_2, written as $u_1 \mid u_2$, first tries to perform the update u_1. If u_1 can be applied, the resulting expression is also the result of $u_1 \mid u_2$. Only if u_1 does not apply, the update u_2 is tried. The *composition* of two updates u_1 and u_2 applies the two updates in that order, even if u_1 can be applied. *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 $\downarrow(\texttt{1+1} \rightsquigarrow \texttt{1})$ applied to `1+(1+1)` results in the expression `1+1`. We use the recursion operator only implicitly in scope updates and do not offer it to the user.

The update operations described thus far do not take into account the scope of identifiers; they are rather like global search-and-replace rules. In contrast to global updates, scope updates always operate only on the uses of a symbol introduced by a particular definition.

In a *scope update*, each element of a definition `let` v `=` d `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 $\{v \rightsquigarrow v' : u_d\}u_u$ for an update that renames v to v', changes v's definition by u_d, and all of its uses by u_u. We call $v \rightsquigarrow v'$ the *name update*, 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 metavariables (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 metavariable). Either one of the variables (but not both) can be missing from the name update. These special

cases describe the creation ($\rightsquigarrow v'$) or removal ($v \rightsquigarrow$) of a binding. In both cases, the definition update is replaced by an expression, which is optional for the binding creation but required in the case of binding removal because it is needed to replace all occurrences of the removed variable. Note that e' must not contain the variable that is to be removed. In the case of binding creation, for example, when applying $\{\rightsquigarrow v = e'\}u_u$ to e, e' is optional and is used, if present, to create an expression `let` $v = e_1$ `in` e_2 where e_1 is the result of applying u_u to e' and e_2 is the result of applying u_u to e. Otherwise, that is, if e' is missing, the result is $\lambda v.e_2$.

At first sight it seems that we also need a combinator to generate fresh variables in order to rename variables and to create new definitions. However, we know exactly all the places of an update where fresh variables are required, namely only in a renaming update or a binding creation, so that we can integrate the generation of fresh variables into the semantics for updates.

The syntax of updates is summarized in Fig. 7.

u	::=	ι	Identity
	|	$p \rightsquigarrow p$	Rule
	|	$\{x \rightsquigarrow x : u\}u$	Change Scope
	|	$\{\rightsquigarrow v[= e]\}u$	Insert Scope
	|	$\{x \rightsquigarrow : e\}u$	Delete Scope
	|	$u \,;\, u$	Composition
	|	$u \mid u$	Alternative
	|	$\downarrow u$	Recursion

Fig. 7. Syntax of updates.

We use an abbreviated notation for scope updates that do not change names, that is, we write $\{v : u_d\}u_u$ instead of $\{v \rightsquigarrow v : u_d\}u_u$. 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 $\{v \rightsquigarrow v' : u_d\}$ and $\{v \rightsquigarrow v'\}u_u$, respectively, and are equivalent to updates $\{v \rightsquigarrow v' : u_d\}\iota$ and $\{v \rightsquigarrow v' : \iota\}u_u$, respectively.

Let us consider some examples. We already have seen examples for rules. A simple example of change scope is an update for consistently renaming variables

```
{v⇝w}v⇝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.

Recall the function generalization update from Section 2. A generalization of a function `f` can be expressed by the following update u.

```
{f:{⇝w}1⇝w}f⇝f 1
```

u is a change-scope update for `f`, which does not rename `f`, but whose definition update (`{⇝w}1⇝w`) introduces a new variable `w` and replaces all occurrences of a particular constant expression (here `1`) by `w` in the definition of `f`. The use update of u, that is, `f ⇝ f 1`, ensures 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. Although we can express a particular function generalization with the update calculus, the definition of a reusable update function like *genFun* requires the extension of the update calculus by abstraction, application, and variables, which is not difficult and which we have omitted here for simplicity.

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 each `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:m ⇝ Int->m}
    ({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, which is a type bound to the metavariable *m*, 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 alternative 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.

5.2.3. Semantics of updates

In the definition of the semantics for alternative updates and recursion we need to know whether an update u is applicable to an expression e, which is the case if the semantics can be used to derive a result expression, that is, $\exists e' : [\![u]\!](e) = e'$. Otherwise, when the semantics gets stuck, we say that u is not applicable and write $[\![u]\!](e) = \bot$. As an abbreviation for the semantics rules we use the update operation "try u" that tries to apply the update u to an expression e and returns the possibly changed expression if u is applicable to e. However, if u is not applicable to e, try u yields e.

Generally, an update u is applied to an expression e recursively matching the structure of u and e. In the procedure of application, there are three issues we need to consider.

First, at the point where a rule is applied to an expression, the rule is applicable only if all the free (object) variables on the left-hand side of the rule are in scope of a enclosing change or delete update. Consider, for example, the rule `f ⇝ f 1`, which is the use update of the function generalization. Here, `f` is a free variable of the rule and is contained in the scope of the current update because the scope update extends the scope by `f`. We denote this set of variables as ρ_S.

We also have to consider the set of variables that are bound by the expression being updated but that are not in scope of the update, because in the semantics definition we have to ensure that newly introduced bound variables are fresh, that is, they must not yet be bound in the updated expression. We denote this set of variables as ρ_B. As an example consider the situation when we apply the function generalization to a function definition for `f` that is local to a function definition for `w`. In this case, it might happen that `f`'s definition contains a reference to `w`. Now if we extended `f`'s definition by a new parameter `w`, all those references would be illegally captured by that parameter and would not refer to the enclosing `w` anymore. To prevent this name capture, we have to ensure that we use `w` only if it is not bound in the current program to be updated, otherwise we have to rename `w` appropriately. We also require that all the free (object) variables of the right-hand side of a rule are contained in the set of bound variables to prevent the creation of unbound variables.

Moreover, consider the case when a change or delete update is applied. If the bound variable is a metavariable, we need to match the metavariable against the object variable in the expression and replace all the occurrences of that metavariable in the update by the object variable. If the bound variable is an object variable, it has to be the same variable in the expression.

We define the semantics in two steps. In the first step, we instantiate a general update based on the expression being applied, checking bound variables, replacing metavariables, and renaming fresh variables. In step two, this update is applied to the expression according to the semantics defined in Fig. 8.

Instantiation takes place in a context of the two aforementioned sets of variables represented by a two-set partition $\rho = (\rho_S, \rho_B)$ where ρ_S contains the variables that are in scope of the update and ρ_B contains the variables that are

bound in the updated expression but that are not in scope of the update. We use two operations for moving variables between ρ_S and ρ_B:

$$(\rho_S, \rho_B)\langle v := (\rho_S \cup \{v\}, \rho_B - \{v\})$$
$$(\rho_S, \rho_B)\rangle v := (\rho_S - \{v\}, \rho_B \cup \{v\})$$

We use the notation $v\rangle^e_\rho w$ to express the fact that w is a variable that is fresh with respect to the expression e and the environment ρ_B. This is a variable that neither is bound in e nor occurs in the current context (ρ_B). If v has this property, $w = v$, otherwise an appropriate name will be constructed (for example, by repeatedly appending a prime symbol until a fresh symbol is found). It is not a problem when v occurs in ρ_S, because in that case the v in ρ_S will be either renamed or deleted. We use the abbreviation $v\langle x \rightsquigarrow x'\rangle^e_\rho w$ that expresses the condition that x matches the bound variable v and the fresh variable generated from x' is w. This predicate formalizes two steps of a scope update: (1) the initial matching of the variable v to which the scope update is applied, and (2) providing a fresh variable, which might be w, to rename w. The predicate is defined as follows.

$$v\langle x \rightsquigarrow x'\rangle^e_\rho w \iff x \gtrsim_\sigma v \wedge \sigma(x')\rangle^e_\rho w$$

The predicate covers two cases: First, if x is a metavariable, say m, we require that if x' is a metavariable, then $x' = x = m$. In this situation we obtain $v\langle m \rightsquigarrow m\rangle^e_\rho v$, which is always satisfied. Second, x is the object variable v and $x' = v'$ (where v' might be v) or $x = m$ and $x' = v'$. Then $v\langle v \rightsquigarrow v'\rangle^e_\rho w$ is satisfied by $v'\rangle^e_\rho = w$.

We also use the notations $[v\langle x]u$ for the update u with all free left occurrences of x substituted by v, $[w\rangle x']u$ for the update u with all free right occurrences of x' substituted by w, and $[v\langle x, w\rangle x']u$ for $[w\rangle x']([v\langle x]u)$.

The judgment $u \xrightarrow[\rho]{e} u'$ denotes that update u is instantiated to u' with respect to the expression e under the context ρ. For a rule update, it is necessary to ensure the free variables are bound:

$$\frac{FV(l) \subseteq \rho_S \qquad FV(r) \subseteq \rho_B}{l \rightsquigarrow r \xrightarrow[\rho]{e} l \rightsquigarrow r}$$

In the case of change updates, metavariables, if any, will be matched and replaced. Newly introduced variables will be renamed if necessary:

$$\frac{v\langle x \rightsquigarrow x'\rangle^{(e\ d)}_\rho w \qquad [v\langle x, w'\rangle x']u_d \xrightarrow[\rho\langle v\rangle w]{(e\ d)} u'_d \qquad [v\langle x, w'\rangle x']u_u \xrightarrow[\rho\langle v\rangle w]{e} u'_u}{\{x \rightsquigarrow x' : u_d\}u_u \xrightarrow[\rho]{\texttt{let } v = d \texttt{ in } e} \{v \rightsquigarrow w : u'_d\}u'_u}$$

Note that the use of $(e\ d)$ in the above rule is to ensure the freshness of w with respect to both e and d, because a `let` expression can bind a recursively defined function. Two more rules are needed for instantiating a change update, which are for beta redex and lambda abstractions. They only differ in that d is not needed in the first two judgments. Instantiation for insert and delete updates are also very similar and omitted. After instantiation, the update is applied based on the rules defined in Fig. 8.

The semantics definition of top-down recursion uses an auxiliary operation μ that applies updates only on recursive occurrences of expressions.

We demonstrate the semantics with a small example. Let $u = \{\texttt{f} : u_d\}u_u$ where $u_d = \{\rightsquigarrow \texttt{w}\}1 \rightsquigarrow \texttt{w}$ and $u_u = \texttt{f} \rightsquigarrow \texttt{f 1}$ and let $e_0 = \texttt{let } d \texttt{ in } e$ where $d = \lambda\texttt{v.v+1}$ and $e = \texttt{f 3}$. We first show that the result of instantiating u with regard to e_0 and an empty context $(\varnothing, \varnothing)$ is u itself. That is,

$$u \xrightarrow[(\varnothing,\varnothing)]{e_0} u$$

This can be inferred from

1. $\texttt{f}\langle\texttt{f} \rightsquigarrow \texttt{f}\rangle^{(e\ d)}_{(\varnothing,\varnothing)}\texttt{f}$
2. $[\texttt{f}\langle\texttt{f}, \texttt{f}\rangle\texttt{f}](\{\rightsquigarrow\texttt{w}\}1 \rightsquigarrow \texttt{w}) = \{\rightsquigarrow\texttt{w}\}1 \rightsquigarrow \texttt{w}$ and $\{\rightsquigarrow\texttt{w}\}1 \rightsquigarrow \texttt{w} \xrightarrow[(\{\texttt{f}\},\{\texttt{f}\})]{e} \{\rightsquigarrow\texttt{w}\}1 \rightsquigarrow \texttt{w}$
3. $[\texttt{f}\langle\texttt{f}, \texttt{f}\rangle\texttt{f}](\texttt{f} \rightsquigarrow \texttt{f 1}) = \texttt{f} \rightsquigarrow \texttt{f 1}$ and $\texttt{f} \rightsquigarrow \texttt{f 1} \xrightarrow[(\{\texttt{f}\},\{\texttt{f}\})]{d} \texttt{f} \rightsquigarrow \texttt{f 1}$

$$\rightsquigarrow_{[\![\,]\!]} \frac{l \gtrsim_{\sigma} e \qquad \sigma(r) = e'}{[\![l \rightsquigarrow r]\!](e) = e'} \qquad \iota_{[\![\,]\!]} \frac{}{[\![\iota]\!](e) = e}$$

$$\{:\}^{chg}_{[\![\,]\!]} \frac{[\![u_d]\!](d) = d' \quad [\![\downarrow u_u]\!](e) = e'}{[\![\{v \rightsquigarrow w\colon u_d\} u_u]\!]((\lambda v.e)\ d) = (\lambda w.e')\ d'} \qquad \frac{[\![\downarrow u_u]\!](e) = e'}{[\![\{v \rightsquigarrow w\colon \iota\} u_u]\!](\lambda v.e) = \lambda w.e'}$$

$$\frac{[\![\downarrow u_u]\!]([\![u_d]\!](d)) = d' \quad [\![\downarrow u_u]\!](e) = e'}{[\![\{v \rightsquigarrow w\colon u_d\} u_u]\!]_{\rho}(\texttt{let } v = d \texttt{ in } e) = \texttt{let } w = d' \texttt{ in } e'}$$

$$\{:\}^{ins}_{[\![\,]\!]} \frac{[\![\downarrow u]\!](e) = e'}{[\![\{\rightsquigarrow v = d\} u]\!](e) = \texttt{let } v = d \texttt{ in } e'} \qquad \frac{[\![\downarrow u]\!](e) = e'}{[\![\{\rightsquigarrow v\} u]\!](e) = \lambda v.e'}$$

$$\{:\}^{del}_{[\![\,]\!]} \frac{[\![\downarrow (u\ ;\ v \rightsquigarrow e_0)]\!](e) = e'}{[\![\{v \rightsquigarrow\colon e_0\} u]\!](\texttt{let } v = d \texttt{ in } e) = e'} \qquad \frac{[\![\downarrow (u\ ;\ v \rightsquigarrow e_0)]\!](e) = e'}{[\![\{v \rightsquigarrow\colon e_0\} u]\!](\lambda v.e) = e'}$$

$$|_{[\![\,]\!]} \frac{[\![u_1]\!](e) = e'}{[\![u_1 \mid u_2]\!](e) = e'} \qquad \frac{[\![u_1]\!](e) = \bot \quad [\![u_2]\!](e) = e'}{[\![u_1 \mid u_2]\!](e) = e'}$$

$$;_{[\![\,]\!]} \frac{[\![\text{try } u_1]\!](e) = e' \quad [\![u_2]\!](e') = e''}{[\![u_1\ ;\ u_2]\!](e) = e''}$$

$$\downarrow_{[\![\,]\!]} \frac{[\![\text{try } u]\!](e) = e' \qquad [\![\mu u]\!](e') = e''}{[\![\downarrow u]\!](e) = e''} \qquad \mu_{[\![\,]\!]} \frac{[\![\text{try } u]\!](v) = e}{[\![\mu u]\!](v) = e} \qquad \frac{[\![\downarrow u]\!](e) = e'}{[\![\mu u]\!](\lambda v.e) = \lambda v.e'}$$

$$\frac{[\![\downarrow u]\!](e_1) = e_1' \qquad [\![\downarrow u]\!]_{\rho}(e_2) = e_2'}{[\![\mu u]\!](e_1\ e_2) = e_1'\ e_2'} \qquad \frac{[\![\downarrow u]\!](d) = d' \qquad [\![\downarrow u]\!](e) = e'}{[\![\mu u]\!](\texttt{let } v = d \texttt{ in } e) = \texttt{let } v = d' \texttt{ in } e'}$$

$$\text{TRY}_{[\![\,]\!]} \frac{[\![u]\!](e) = e'}{[\![\text{try } u]\!](e) = e'} \qquad \frac{[\![u]\!](e) = \bot}{[\![\text{try } u]\!](e) = e}$$

Fig. 8. Semantics of updates.

We then show that

$$[\![u]\!](e_0) = \texttt{let f=}\lambda\texttt{w.}\lambda\texttt{v.v+w in f 1 3}$$

This can be obtained by applying $\{:\}^{chg}_{[\![\,]\!]}$ and the following premises.

4. $[\![\downarrow u_u]\!]([\![\downarrow u_d]\!](\lambda\texttt{v.v+1})) = \lambda\texttt{w.}\lambda\texttt{v.v+w}$
5. $[\![\downarrow u_u]\!](\texttt{f 3}) = \texttt{f 1 3}$

(4) can be further obtained by showing:

6. $[\![u_d]\!](\lambda\texttt{v.v+1}) = \lambda\texttt{w.}\lambda\texttt{v.v+w}$
7. $[\![u_u]\!](\lambda\texttt{w.}\lambda\texttt{v.v+w}) = \lambda\texttt{w.}\lambda\texttt{v.v+w}$

(6) is obtained by applying $\{:\}^{ins}_{[\![\,]\!]}$ with the following premises.

8. $[\![\downarrow 1 \rightsquigarrow \texttt{w}]\!](\lambda\texttt{v.v+1}) = \lambda\texttt{v.v+w}$

(8) can further be obtained by applying $\downarrow_{[\![\,]\!]}$ to the following premises:

9. $[\![\texttt{try}\ (1 \rightsquigarrow \texttt{w})]\!](\lambda\texttt{v.v+1}) = \lambda\texttt{v.v+1}$
10. $[\![\mu(1 \rightsquigarrow \texttt{w})]\!](\lambda\texttt{v.v+1}) = \lambda\texttt{v.v+w}$

(9) is a simple application of $\text{TRY}_{[\![\,]\!]}$ and $\rightsquigarrow_{[\![\,]\!]}$. (10) can be obtained by the repeatedly applying the congruence rules of $\mu_{[\![\,]\!]}$ and $\downarrow_{[\![\,]\!]}$.

The goal of the update calculus is to provide a means for updating programs without introducing type errors. However, there is no practical way to analyze the semantics and logic of the object program. Logic errors might be introduced in the updated program, and it is the programmer's responsibility to manually change the updated program to eliminate such logic errors. For example, in the update introduced in Section 2, `succ s` is introduced as a new first argument to constructor `Node`, because this update is intended to be applied to the `insert` function that increases the size of the node by 1. However, imagine an `insert` function that does not insert duplicate elements. In this case, the size of the node does not always increase, so that some `Node` expressions might receive incorrect size values after the update.

5.3. A type system for the update calculus

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.

5.3.1. 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 ($t \rightsquigarrow t$), but it can also be an alternative of type changes ($\delta|\delta$). For example, the type change of the update $1 \rightsquigarrow \texttt{True}$ is $\texttt{Int} \rightsquigarrow \texttt{Bool}$, while the type change of $1 \rightsquigarrow \texttt{True} \mid \texttt{odd} \rightsquigarrow 2$ is $\texttt{Int} \rightsquigarrow \texttt{True}|\texttt{Int->Bool} \rightsquigarrow \texttt{Int}$.

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 subexpression 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.

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

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 $u' = 1 \rightsquigarrow \texttt{w}$ is $\texttt{Int} \rightsquigarrow \texttt{a}$. However, when u' is used as a use update of a scope update $u = \{\rightsquigarrow\texttt{w}\}1 \rightsquigarrow \texttt{w}$, it is performed recursively, so that the type change is described using a context type $C\langle\texttt{Int}\rangle \rightsquigarrow C\langle\texttt{a}\rangle$.

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 `λw.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

δ	::=	$\tau \rightsquigarrow \tau \mid \delta \mid \delta$	Type change
τ	::=	$t \mid \tau \to \tau \mid C\langle\tau\rangle \mid \tau_{\mid C}$	Extended type
C	::=	$\epsilon \mid \hookrightarrow t \mid t \hookrightarrow t$	Type hook

Fig. 9. Type changes.

on a type variable that might replace e. Such a *constrained type* is written as $\mathtt{a}_{|C}$. Its meaning is to constrain a type variable a by the type of a subexpression (represented by the left part of a type hook):

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

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

To see how type hooks, context types, and constrained types work, consider the application of u to `1`, which yields `λw.w`. The corresponding type change `Int` $\rightsquigarrow$ `a->a` is obtained using the type hook ϵ. However, applied to `odd 1`, u yields `λw.odd w` with the type change `Bool` $\rightsquigarrow$ `Int->Bool`, which is obtained from the type hook `Int`$\hookrightarrow$`Bool`. As another example consider the renaming update $u = \{\mathtt{v} \rightsquigarrow \mathtt{w}\}\mathtt{v} \rightsquigarrow \mathtt{w}$. For the update we obtain a type change $C\langle\mathtt{a}_{|C}\rangle \rightsquigarrow C\langle\mathtt{b}_{|C}\rangle$, which is the same as $C\langle\mathtt{a}_{|C}\rangle \rightsquigarrow C\langle\mathtt{a}_{|C}\rangle$ because the scopes of the two type variables are distinct. The type hook C results for the same reason as in the previous example. Applying u to the expression `λv.1` yields `λw.1` with a type change `a->Int` $\rightsquigarrow$ `a->Int`, which is obtained by using the type hook $\hookrightarrow$`a->Int`. Similarly, `λv.v` is mapped by u into `λw.w` with a type change `a->a` $\rightsquigarrow$ `a->a`, which is an instance of u's type change for the context ϵ. Finally, u changes `λv.odd v` to `λw.odd w` with a type change `Int->Bool` $\rightsquigarrow$ `Int->Bool`. This type change is u's type change specialized for the type hook `Int`$\hookrightarrow$`Int->Bool`.

The syntax of hooks, contexts, and type changes is defined in Fig. 9. To summarize, a type change can be given by a pair of simple types $t \rightsquigarrow t$ or by a pair of the more general context or constrained types ("extended types"), that is, $\tau \rightsquigarrow \tau$. A type change can also be given by an alternative of two type changes $\delta|\delta$.

To explain the meaning of nested contexts, we define the composition of type hooks as follows.

$$\begin{aligned} \epsilon \cdot C &:= C \\ C \cdot \epsilon &:= C \\ t_1 \hookrightarrow t_2 \cdot t_1' \hookrightarrow t_2' &:= \begin{cases} t_1' \hookrightarrow t_2 & \text{if } t_2' \succ t_1 \vee t_1 \text{ is empty} \\ \text{error} & \text{otherwise} \end{cases} \end{aligned}$$

It is easy to verify that with this definition we obtain the following two equalities.

$$\begin{aligned} C \cdot C'\langle t\rangle &= C\langle C'\langle t\rangle\rangle \\ t_{|C \cdot C'} &= (t_{|C})_{|C'} \end{aligned}$$

The effect is that the outermost hooks affect context types, whereas the innermost hooks are relevant for constrained types. Finally, since the inference rules generate, in general, context constraints for arbitrary type changes, we have to explain how contexts extend to type changes and alternative type changes.

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

The definitions of free variables and generic instances extend naturally to type hooks, context types, and constrained types.

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 $t \vec{\prec} t' \to t$. The rationale for this definition is that two updates u and u' of different type changes $t_1 \rightsquigarrow t_2$ and $t_1' \rightsquigarrow t_2'$, respectively, can be considered well typed in an alternative $u \mid u'$ if one type change is an applicative instance of the other, that is, if $t_1 \rightsquigarrow t_2 \vec{\prec} t_1' \rightsquigarrow t_2'$ or $t_1' \rightsquigarrow t_2' \vec{\prec} t_1 \rightsquigarrow t_2$, because in that case one update is just more specific than the other. Consider, for example, the following update.

`{f:succ` $\rightsquigarrow$ `plus}f` $m \rightsquigarrow$ `f` m `1 | f` $\rightsquigarrow$ `f 1`

The first rule of the alternative `f m ⇝ f m 1` has the type change `Int ⇝ Int` whereas the second rule `f ⇝ f 1` has the type change `Int->Int⇝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 `Int ⇝ Int` being an applicative instance of `Int->Int ⇝ Int->Int`. The relationship is defined in Fig. 10.

$$\mathrm{REFL}_{\overrightarrow{\prec}}\ \frac{}{\tau \overrightarrow{\prec} \tau} \qquad C_{\overrightarrow{\prec}}\ \frac{\tau \overrightarrow{\prec} \tau'}{C\langle\tau\rangle \overrightarrow{\prec} C\langle\tau'\rangle} \qquad \frac{\tau \overrightarrow{\prec} \tau'}{\tau_{|C} \overrightarrow{\prec} \tau'_{|C}} \qquad \rightarrow_{\overrightarrow{\prec}}\ \frac{\tau \overrightarrow{\prec} \tau_2}{\tau \overrightarrow{\prec} \tau_1 \to \tau_2}$$

$$\rightsquigarrow_{\overrightarrow{\prec}}\ \frac{\tau_1 \overrightarrow{\prec} \tau'_1 \qquad \tau_2 \overrightarrow{\prec} \tau'_2}{\tau_1 \rightsquigarrow \tau_2 \overrightarrow{\prec} \tau'_1 \rightsquigarrow \tau'_2} \qquad |_{\overrightarrow{\prec}}\ \frac{\tau_1 \rightsquigarrow \tau_2 \overrightarrow{\prec} \tau'_1 \rightsquigarrow \tau'_2 \qquad \delta \overrightarrow{\prec} \delta'}{\tau_1 \rightsquigarrow \tau_2 | \delta \overrightarrow{\prec} \tau'_1 \rightsquigarrow \tau'_2 | \delta'}$$

Fig. 10. Applicative instance.

5.3.2. Type-change inference

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

(1) $x \rightsquigarrow x' :: \tau \rightsquigarrow \tau'$ expresses that x of type τ is changed to x' of type τ'. The following constraint applies: if x' is a metavariable, then $x' = x$ and $\tau' = \tau$.
(2) $v :_r \tau$ expresses that v is a newly introduced (object) variable of type τ.
(3) $x :_\ell \tau$ expresses that x is an (object or meta) variable of type τ 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. In computing these projections we have to map extended types τ to simple types t, which is performed by dropping potential hook variables and constraints as follows.

$$\begin{array}{lcl} \lfloor t \rfloor & = & t \\ \lfloor C\langle\tau\rangle \rfloor & = & \lfloor \tau \rfloor \\ \lfloor \tau_{|C} \rfloor & = & \lfloor \tau \rfloor \\ \lfloor \tau_1 \to \tau_2 \rfloor & = & \lfloor \tau_1 \rfloor \to \lfloor \tau_2 \rfloor \end{array}$$

Now the projections are defined as follows:

$$\Delta_\ell := \{x : \lfloor \tau \rfloor \mid x \rightsquigarrow x' :: \tau \rightsquigarrow \tau' \in \Delta\} \cup \{x : \lfloor \tau \rfloor \mid x :_\ell \tau \in \Delta\}$$
$$\Delta_r := \{x' : \lfloor \tau' \rfloor \mid x \rightsquigarrow x' :: \tau \rightsquigarrow \tau' \in \Delta\} \cup \{x' : \lfloor \tau' \rfloor \mid x' :_r \tau' \in \Delta\}$$

The type-change rules are defined in Fig. 11. The rules for creating or deleting a binding have to insert a function argument type on either the right or the left part of a type change. This type insertion works across alternative type changes; we use the notation $\tau \underset{\ell}{\to} \delta$ ($\tau \underset{r}{\to} \delta$) to extend the argument (result) type of a type change to a function type. The definition is as follows.

$$\begin{array}{l} \tau \underset{\ell}{\to} (\tau_l \rightsquigarrow \tau_r) := (\tau \to \tau_l) \rightsquigarrow \tau_r \\ \tau \underset{r}{\to} (\tau_l \rightsquigarrow \tau_r) := \tau_l \rightsquigarrow (\tau \to \tau_r) \\ \tau \underset{\ell}{\to} (\delta | \delta') := (\tau \underset{\ell}{\to} \delta) | (\tau \underset{\ell}{\to} \delta') \\ \tau \underset{r}{\to} (\delta | \delta') := (\tau \underset{r}{\to} \delta) | (\tau \underset{r}{\to} \delta') \end{array}$$

The inference rule $\rightsquigarrow_\triangleright$ connects the type system of the underlying object language (lambda calculus) with the type-change system. This rule is rather simple since rule updates cannot contain binding constructs, which means that all variables used in either e or e' have to be brought into Δ by scope updates.

We demonstrate the type-change rules by a small example. Again, let $u =$ `{f:{⇝w}1⇝w}f⇝f 1`. We will now show that

$$\varnothing \triangleright u :: C\langle \texttt{Int} \rightsquigarrow \texttt{Int} \rangle$$

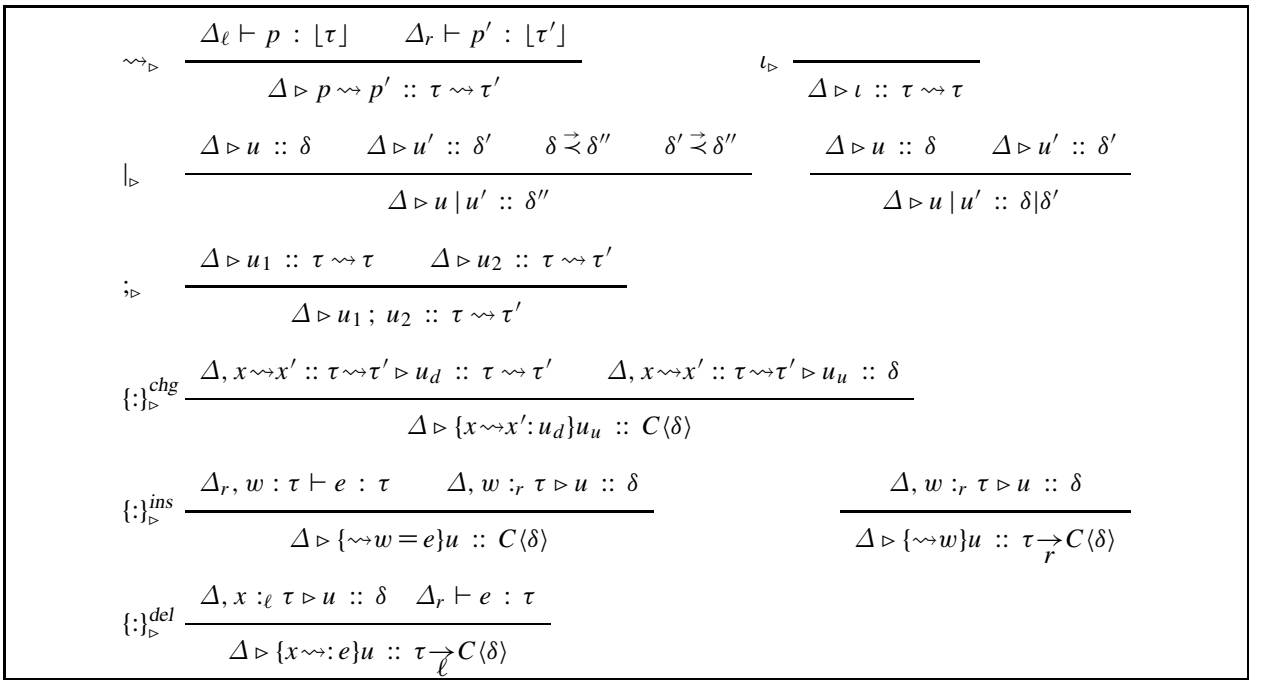

Fig. 11. Type-change system.

This type change can be inferred from $\{:\}^{chg}_{\triangleright}$ and the following premises:

1. $\texttt{f} \rightsquigarrow \texttt{f} :: C\langle\texttt{Int}\rangle \rightsquigarrow \texttt{Int} \rightarrow C\langle\texttt{Int}\rangle \triangleright \{\rightsquigarrow \texttt{w}\}\texttt{1} \rightsquigarrow \texttt{w} :: C\langle\texttt{Int}\rangle \rightsquigarrow \texttt{Int} \rightarrow C\langle\texttt{Int}\rangle$
2. $\texttt{f} \rightsquigarrow \texttt{f} :: C\langle\texttt{Int}\rangle \rightsquigarrow \texttt{Int} \rightarrow C\langle\texttt{Int}\rangle \triangleright \texttt{f} \rightsquigarrow \texttt{f 1} :: \texttt{Int} \rightsquigarrow \texttt{Int}$

(2) results from an application of $\rightsquigarrow_{\triangleright}$, and (1) can be obtained by applying $\{:\}^{ins}_{\triangleright}$ and the following premise:

3. $\texttt{w} :_r \texttt{Int} \triangleright \texttt{1} \rightsquigarrow \texttt{w} :: \texttt{Int} \rightsquigarrow \texttt{Int}$

which is, again, simply an application of $\rightsquigarrow_{\triangleright}$.

One major limitation to the update calculus is the composition. It is not practical to foresee the object program when we analyze the updates statically. Therefore we do not know how the first update will change the program. Hence, in order to guarantee the safety of the update, we require that the first update in a composed update does not have a type change.

5.3.3. 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* (see Section 4). 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 $\tau \rightsquigarrow \tau' | \delta$ we require that $\lfloor\tau\rfloor$ is a generic instance of $\lfloor\tau'\rfloor$ or that one extended type, τ or τ', 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. We can find out about possible type errors by looking at the type system for lambda calculus (see Fig. 5): Essentially, type inference fails either in the rule $\text{VAR}_{\vdash}$ if the type for a variable cannot be found in the type environment or in the rule $\text{APP}_{\vdash}$ if the parameter type of a function does not agree with the type of the argument to which it is applied. On the other hand, the rules $\text{ABS}_{\vdash}$ and $\text{LET}_{\vdash}$ eventually refer to $\text{VAR}_{\vdash}$ and $\text{APP}_{\vdash}$ to ensure typing constraints that might fail. Let us now consider how updates can possibly introduce these errors.

Unbound variables. The free-variable problem can be introduced into an expression by an update that renames a bound variable without accordingly renaming all references to that variables; unrenamed variables might become free, thus leading to an error in the $\text{VAR}_{\vdash}$ rule, or they might be bound by an enclosing λ or `let` and thus might change the type that is obtained by the $\text{VAR}_{\vdash}$ rule, thus breaking eventually the $\text{APP}_{\vdash}$ rule. These kinds of changes are prevented by condition (A) and the type-change rules. Free variables could also be introduced by rules, such as $\texttt{1} \rightsquigarrow \texttt{x}$ (where `x` is not bound). However, these kinds of updates are prevented by the type-change system since we cannot derive a type change for $\texttt{1} \rightsquigarrow \texttt{x}$ when we have no assumption about `x` in the type-change environment.

Incorrect application. An application can become ill typed if the type of the function or the argument changes without a corresponding change of the other part of the application. Types can be changed by rules that replace objects of one type by objects of another type as in $\texttt{1} \rightsquigarrow \texttt{True}$. Such a change is only problematic if it is applied to a subexpression; otherwise, a rule cannot change only one part of an application. However, since application to subexpressions can only happen through the recursion in use updates, such an update is not possible since it violates the condition (B) (in the example: `Int` is not an instance of `Bool`). Types can also be changed by change-scope updates. By just changing the type of a variable, say `v` from τ to τ', we can break the type correctness in two different ways: applications of `v` as well as contexts of `v` (that is, applications of other expressions to `v`) can become ill typed. Both cases are prevented by conditions (A) and (B) together with the type-change system, because (i) having an update $u = \texttt{v} \rightsquigarrow \texttt{e}$ ensures that all occurrences of `v` are changed (not necessarily by this rule, but no occurrence of `v` is left unchanged), and (ii) the type change $\tau \rightsquigarrow \tau'$ derived by the type system is required to be "generalizing" (that is, $\tau' \succ \tau$, see below), which ensures that `e` fits all contexts typewise.

Let us now express the well-structuring constraint more formally. We first identify some properties of change-scope updates. Let $u = \{x \rightsquigarrow x' : u_d\}u_u$ and let $x \rightsquigarrow x' :: \tau_{|C} \rightsquigarrow \tau'_{|C}$ be the assumption that has been used in rule $\{:\}^{chg}_{\triangleright}$ to derive its type change, say $C\langle\tau_1 \rightsquigarrow \tau_2 | \delta\rangle$.

(1) u is *self-contained* iff $x \neq x' \vee \tau \neq \tau' \implies \exists u', u'', p$ such that $u_u = u' \mid x \rightsquigarrow p \mid u''$.
(2) u is *smooth* iff $\tau' \succ \tau$ or $\tau \overset{\rightarrow}{\prec} \tau'$ or $\tau' \overset{\rightarrow}{\prec} \tau$.
(3) u is (at most) *generalizing* iff $\tau_2 \succ \tau_1$.

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 express this result in the following theorem.

For the theorem, we need the notion of *type-conform applicability*. A change-scope update $u = \{x \rightsquigarrow x' : u_d\}u_u$ or a delete-scope update $u = \{x \rightsquigarrow : e_0\}u_u$, is type-conform applicable to an expression $e = (\lambda v.e')\ d$ or $e = \texttt{let}\ v = d\ \texttt{in}\ e'$ if

(a) u is applicable to e
(b) $\downarrow u_u$ is type-conform applicable to e',
(c) u_d is type-conform applicable to d, and
(d) $\Gamma, v : t \triangleright e' :: t' \wedge \Gamma \vdash d : t \implies \tau_{|C} = t$ where $\tau_{|C}$ refers to the type of x used when the type change of u is inferred, that is, it is the same $\tau_{|C}$ as in the rules $\{:\}^{chg}_{\triangleright}$ and $\{:\}^{del}_{\triangleright}$ from Fig. 11.

Other updates u are type-conform applicable to e if they are applicable to e.

Theorem 1 (*Soundness*). *If u is well structured and type-conform applicable to e, then:*

$$\Delta \triangleright u :: C\langle\tau_1 \rightsquigarrow \tau_2 | \delta\rangle \wedge \Delta_\ell \vdash e : t_1 \implies \Delta_r \vdash [\![u]\!](e) : t_2$$

where $t_2 = C\langle\tau_2\rangle$ *for some context* $C\langle\rangle$.

Proof. The proof is by induction over the structure of e and u. First, consider the case $e = (\lambda v.e_0)\ d$ and $u = \{x \rightsquigarrow x' : u_d\}u_u$. We can assume

$$\Delta \triangleright u \ :: \ C\langle\tau_1 \rightsquigarrow \tau_2|\delta\rangle \text{ and} \tag{1}$$
$$\Delta_\ell \vdash e \ : \ C\langle\tau_1\rangle. \tag{2}$$

From (2) and rule $\text{APP}_\vdash$ from the object-language type system it follows that

$$\Delta_\ell \vdash d \ : \ t \quad \text{and} \tag{T1}$$
$$\Delta_\ell \vdash \lambda v.e_0 \ : \ t \to C\langle\tau_1\rangle \tag{T2}$$

for some type t. Next, the first rule $\{:\}^{chg}_{[\![\,]\!]}$ from the semantics of updates tells that $[\![u]\!]_\rho(e) = (\lambda v.e')\ d'$ where

$$v\langle x \rightsquigarrow x'\rangle^e_\rho w \tag{M}$$
$$[\![[v\langle x, w\rangle x']u_d]\!]_\rho(d) = d', \quad \text{and} \tag{U1}$$
$$[\![\downarrow[v\langle x, w\rangle x']u_u]\!]_{\rho\langle v\rangle w}(e_0) = e' \tag{U2}$$

From (1) and the type-change rule $\{:\}^{chg}_{\triangleright}$ we know that

$$\Delta, x \rightsquigarrow x' :: \tau_{|C} \rightsquigarrow \tau'_{|C} \triangleright u_d \ :: \ \tau_{|C} \rightsquigarrow \tau'_{|C}, \quad \text{and} \tag{C1}$$
$$\Delta, x \rightsquigarrow x' :: \tau_{|C} \rightsquigarrow \tau'_{|C} \triangleright u_u \ :: \ \tau_1 \rightsquigarrow \tau_2|\delta. \tag{C2}$$

where we can substitute v for x and w for x' due to (M). Since u is type-conform applicable to e, we know that $\tau_{|C} = t$. Therefore, we can conclude from (C1), (T1), (U1), and the induction hypothesis

$$\Delta_r, w : \tau'_{|C} \vdash d' \ : \ \tau'_{|C}$$

Since $w \notin d'$, we also have:

$$\Delta_r \vdash d' \ : \ \tau'_{|C} \tag{*}$$

Now let us consider u_u. First of all, from (M) and (C2) we can conclude

$$\Delta, v \rightsquigarrow w :: \tau_{|C} \rightsquigarrow \tau'_{|C} \triangleright \downarrow u_u \ :: \ C\langle\tau_1 \rightsquigarrow \tau_2|\delta\rangle \tag{C3}$$

From (T2) and rule $\text{ABS}_\vdash$ we know that

$$\Delta_\ell, v : t \vdash e_0 \ : \ C\langle\tau_1\rangle$$

Since $\downarrow u_u$ is well structured and type-conform applicable to e_0, we can apply the theorem inductively and obtain

$$\Delta_r, w : \tau'_{|C} \vdash e' \ : \ C\langle\tau_2\rangle$$

which is the same as

$$\Delta_r \vdash \lambda w.e' \ : \ \tau'_{|C} \to C\langle\tau_2\rangle$$

Together with (*) it follows from rule $\text{APP}_\vdash$ that

$$\Delta_r \vdash (\lambda w.e')\ d' \ : \ C\langle\tau_2\rangle$$

which proves this case.

Next, consider the case $e = \texttt{let}\ v = d\ \texttt{in}\ e_0$ and $u = \{v \rightsquigarrow w : u_d\}u_u$, which is very similar to the previous case, but slightly more involved due the possible recursion. We can assume

$$\Delta \triangleright u \ :: \ C\langle\tau_1 \rightsquigarrow \tau_2|\delta\rangle \text{ and} \tag{1}$$
$$\Delta_\ell \vdash e \ : \ t_1. \tag{2}$$

From (2) and rule LET$_\vdash$ of the object-language type system it follows that

$$\Delta_\ell, v : t \vdash d \ : \ t \text{ and} \tag{T1}$$
$$\Delta_\ell, v : \forall \bar{a}.t \vdash e_0 \ : \ t_1 \tag{T2}$$

for some type t. Next, the last $\{:\}^{chg}_{[\![\,]\!]}$ rule from the semantics of updates tells that $[\![u]\!](e) = (\lambda v.e')\ d'$ where

$$[\![\downarrow u_u]\!]([\![u_d]\!](d)) = d', \quad \text{and} \tag{U1}$$
$$[\![\downarrow u_u]\!](e_0) = e' \tag{U2}$$

Effectively, $[\![\downarrow u_u]\!]([\![u_d]\!](d))$ is equivalent to $[\![\downarrow u_u\ ;\ u_d]\!](d)$ according to the semantics definition. It can be easily verified against the definitions that $[\![\downarrow u_u\ ;\ u_d]\!](d)$ is well structured since u is well structured and it is type-conform applicable to d.

From (1) and the type-change rule $\{:\}^{chg}_{\triangleright}$ we know that

$$\Delta, v{\rightsquigarrow}w :: \tau_{|C}{\rightsquigarrow}\tau'_{|C} \triangleright u_d \ :: \ \tau_{|C} \rightsquigarrow \tau'_{|C}, \text{ and} \tag{C1}$$
$$\Delta, v{\rightsquigarrow}w :: \tau_{|C}{\rightsquigarrow}\tau'_{|C} \triangleright u_u \ :: \ \tau_1{\rightsquigarrow}\tau_2|\delta. \tag{C2}$$

Since u is type-conform applicable to e, we know that $\tau_{|C} = t$. Therefore, we can conclude from (C1), (T1), (U1), and the induction hypothesis

$$\Delta_r, w : \tau'_{|C} \vdash d' \ : \ \tau'_{|C} \tag{*}$$

Now let us consider u_u. First, from (M), (C2) and type-change rule $\downarrow_\triangleright$, we can conclude

$$\Delta, v{\rightsquigarrow}w :: \tau_{|C}{\rightsquigarrow}\tau'_{|C} \triangleright \downarrow u_u \ :: \ C\langle\tau_1{\rightsquigarrow}\tau_2|\delta\rangle \tag{C3}$$

From (T2) and rule LET$_\vdash$ we know that

$$\Delta_\ell, v : t \vdash e_0 \ : \ t_1$$

Since $\downarrow u_u$ is well structured and type-conform applicable to e_0, we can apply the theorem inductively and obtain

$$\Delta_r, w : \tau'_{|C} \vdash e' \ : \ t_2$$

by rule LET$_\vdash$, together with (*), we can conclude that

$$\Delta_r \vdash \texttt{let}\ w = d'\ \texttt{in}\ e' \ : \ t_2$$

which proves this case. Other cases can be proved similarly. □

Theorem 1 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 theorem 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.

Let us consider the safety of some of the presented example updates. The function generalization update from Section 5.2.2 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). The lambda calculus updates are safe; however, the updates might leave some functions that use expressions of the `Lam` data type unexpanded. Finally, the second size update is safe for programs that only use `Node` expressions in a function `insert`. Similarly, programs that use `Lam` expressions only in a function `eval`, the lambda calculus updates do not cause non-exhaustive `case` expressions. We discuss this aspect briefly in the next section when we talk about extensions of the system.

6. Future work

There are several directions for future work. Regarding the presented calculus, we believe that the following aspects are most promising with respect to extending its expressiveness and usefulness.

Named type changes. Currently, the type-change system only reports a possible change for the result expression. For larger updates it would be interesting to obtain, for example, renamings and type changes for all (or at least all non-local) definitions. This extension seems to be orthogonal to the current system and not difficult to realize.

Conditional update safety. The well-structuring conditions that are required to guarantee safety of updates are rather strict so that some useful program updates would not be classified as safe. However, in many situations, "complete" safety is not mandatory. Instead, a form of conditional safety is sufficient. For example, the second size update and the lambda calculus update could be considered to be *conditionally safe* in the sense that type safety is preserved for those object programs that satisfy some constraints (such as referring to non-globally updated objects only in restricted places). This property is not as strong as unconditional safety, but it is more widely applicable and is still much better than having no information at all.

Allowing non-generalizing type changes. Currently, well-structured updates can change symbols only to more general types. This restriction is required to ensure type correctness because other type changes can break the well typing of contexts, which could only be captured by requiring an update that applies to all possible contexts and basically means to insert expressions that revert the type change. However, with the concept of conditional update safety we might be able to relax the covering criterion.

7. Conclusions

We have argued for a structured approach to perform software changes to help improve the reliability of software maintenance. To this end, we have proposed to perform software updates by programs in an update language that offers type-safety preservation as a criterion that can be statically enforced. In particular, 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 and well-structured updates preserve the well typing of lambda calculus expressions. The presented calculus can serve as the basis for type-safe update languages.

References

[1] ACM, Communications of the ACM 44 (10) (2001).
[2] G. Alkhatib, The maintenance problem of application software: An empirical analysis, Journal of Software Maintenance: Research and Practice 1 (1992) 83–104.
[3] M.W. Bauer (Ed.), Resistance to New Technology: Nuclear Power, Information Technology, and Biotechnology, Cambridge University Press, Cambridge, NY, 1997.
[4] L. Bergmans, M. Askit, Composing crosscutting concerns using composition filters, Communications of the ACM 44 (10) (2001) 51–57.
[5] L. Bernstein, Tidbits, ACM SIGSOFT Software Engineering Notes 18 (3) (1993) 1–55.
[6] N. Bjørner, Type checking meta programs, in: Workshop on Logical Frameworks and Meta-Languages, 1999.
[7] B. Borovanský, C. Kirchner, H. Kirchner, P.E. Moreau, C. Ringeissen, Rewriting with strategies in ELAN: A functional semantics, International Journal of Foundations of Computer Science 12 (1) (2001) 69–95.
[8] B. Borovanský, C. Kirchner, H. Kirchner, P. E. Moreau, M. Vittek, ELAN: A logical framework based on computational systems, in: Workshop on Rewriting Logic and Applications, 1996.
[9] P. Borras, D. Clèment, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, Centaur: The system, in: 3rd ACM SIGSOFT Symp. on Software Development Environments, 1988, pp. 14–24.
[10] B.M. Bouldin (Ed.), Agents of Change: Managing the Introduction of Automated Tools, Yourdon Press, Englewood Cliffs, NJ, 1989.
[11] T. Elrad, M. Askit, G. Kiczales, K. Lieberherr, H. Ossher, Discussing aspects of AOP, Communications of the ACM 44 (10) (2001) 33–39.
[12] M. Erwig, Programs are abstract data types, in: 16th IEEE Int. Conf. on Automated Software Engineering, 2001, pp. 400–403.
[13] M. Erwig, D. Ren, A rule-based language for programming software updates, in: 3rd ACM SIGPLAN Workshop on Rule-Based Programming, 2002, pp. 67–77.
[14] M. Erwig, D. Ren, Type-safe update programming, in: 12th European Symp. on Programming, in: LNCS, vol. 2618, 2003, pp. 269–283.
[15] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Reading, MA, 1999.
[16] M. Hanna, Maintenance burden begging for a remedy, Datamation (April) (1993) 53–63.
[17] S. Horwitz, J. Prins, T. Reps, Integrating non-interfering versions of programs, ACM Transactions on Programming Languages and Systems 11 (3) (1989) 345–387.
[18] W.A. Howard, The formulae-as-types notion of construction, in: J.P. Seldin, J.R. Hindley (Eds.), To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, 1980, pp. 479–490.
[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, Getting started with AspectJ, Communications of the ACM 44 (10) (2001) 59–65.
[20] P. Klint, A meta-environment for generating programming environments, ACM Transactions on Software Engineering and Methodology 2 (2) (1993) 176–201.

[21] P. Klint, R. Lämmel, C. Verhoef, Towards an engineering discipline for grammarware, ACM Transactions on Software Engineering and Methodology 14 (3) (2005) 331–380.
[22] R. Lämmel, Evolution of rule-based programs, Journal of Logic and Algebraic Programming 60–61 (2004) 143–193.
[23] R. Lämmel, S. Peyton Jones, Scrap your boilerplate: A practical design pattern for generic programming, in: ACM SIGPLAN Workshop on Types in Language Design and Implementation, 2003, pp. 26–37.
[24] R. Lämmel, S. Peyton Jones, Scrap more boilerplate: Reflection, zips, and generalised casts, in: ACM Int. Conf. on Functional Programming, 2004, pp. 244–255.
[25] R. Lämmel, J. Visser, Typed combinators for generic traversal, in: 4th Symp. on Practical Aspects of Declarative Languages, in: LNCS, vol. 2257, 2002, pp. 137–154.
[26] R. Lämmel, J. Visser, A Strafunski application letter, in: 5th Symp. on Practical Aspects of Declarative Languages, in: LNCS, vol. 2562, 2003, pp. 357–375.
[27] M.M. Lehman, Program Evolution, Academic Press, London, UK, 1985.
[28] B.P. Lientz, E.B. Swanson, Software Maintenance Management, Addison-Wesley, Reading, MA, 1980.
[29] L. Magnusson, B. Nordström, The ALF proof editor and its proof engine, in: Types for Proofs and Programs, in: LNCS, vol. 806, 1994, pp. 213–237.
[30] P. Martin-Löf, Constructive mathematics and computer programming, in: 6th Int. Congress for Logic, Methodology and Philosophy of Science, 1979, pp. 153–175.
[31] H. Ossher, P. Tarr, Using multidimensional separation of concerns to (Re)shape evolving software, Communications of the ACM 44 (10) (2001) 43–50.
[32] F. Pfenning, Logical frameworks, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Elsevier Science Publishers, Amsterdam, NL, 2001 (Chapter 21).
[33] L. Prechelt, W.F. Tichy, A controlled experiment to assess the benefits of procedure argument type checking, IEEE Transactions on Software Engineering 24 (4) (1998) 302–312.
[34] R. Pressman, Software Engineering: A Practitioner's Approach, 5th ed., McGraw-Hill, New York, NY, 2001.
[35] T. Reps, Algebraic properties of program integration, Science of Computer Programming 17 (1991) 139–215.
[36] T.W. Reps, T. Teitelbaum, The Synthesizer Generator: A System for Constructing Language-Based Editors, Springer-Verlag, New York, 1989.
[37] L. Rideau, L. Thèry, An interactive programming environment for ML, Rapport de Recherche 3139, INRIA, Sophia Antipolis, 1997.
[38] D. Roberts, J. Brant, Refactoring Tools, in: M. Fowler (Ed.), Refactoring: Improving the Design of Existing Code, Addison-Wesley, Reading, MA, 1999, pp. 309–352 (Chapter 14).
[39] T. Sheard, Accomplishments and research challenges in meta-programming, in: 2nd Int. Workshop on Semantics, Applications, and Implementation of Program Generation, in: LNCS, vol. 2196, 2001, pp. 2–44.
[40] T. Sheard, S.L. Peyton Jones, Template metaprogramming for Haskell, in: Haskell Workshop, 2002, pp. 1–16.
[41] W. Taha, T. Sheard, MetaML and multi-stage programming with explicit annotations, Theoretical Computer Science 248 (1–2) (2000) 211–242.
[42] A.A. Takang, P.A. Grubb, Software Maintenance: Concepts and Practice, Thomson Computer Press, London, UK, 1996.
[43] M.G.J. van den Brand, P. Klint, J.J. Vinju, Term rewriting with traversal functions, ACM Transactions on Software Engineering and Methodology 12 (2) (2003) 152–190.
[44] C. Verhoef, How to implement the future, in: 26th Euromicro Conference, 2000, pp. 32–47.
[45] E. Visser, Strategic pattern matching, in: 10th Int. Conf. on Rewriting Techniques and Applications, in: LNCS, vol. 1631, 1999, pp. 30–44.
[46] E. Visser, Language independent traversals for program transformation, in: Workshop on Generic Programming, 2000, Technical Report UU-CS-2000-19, Universiteit Utrecht.
[47] E. Visser, Stratego: A language for program transformation based on rewriting strategies, in: 12th Int. Conf. on Rewriting Techniques and Applications, in: LNCS, vol. 2051, 2001.
[48] E. Visser, Z. Benaissa, A. Tolmach, Building program optimizers with rewriting strategies, in: 3rd ACM Int. Conf. on Functional Programming, 1998, pp. 13–26.
[49] E. Visser, et al., The online survey of program transformation, http://www.program-transformation.org/survey.html.
[50] J. Whittle, A. Bundy, R. Boulton, H. Lowe, An ML editor based on proof-as-programs, in: 9th Int. Symp. on Programming Language Implementation and Logic Programming, in: LNCS, vol. 1292, 1997, pp. 389–405.
[51] J. Whittle, A. Bundy, H. Lowe, An editor for helping novices to learn standard ML, in: 14th Int. Conf. on Automated Software Engineering, 1999.