An Update Calculus for Expressing Type-Safe Program Updates*

Martin Erwig and Deling Ren

Oregon State University
Department of Computer Science
Corvallis, OR 97331, USA
[erwig|rende]@cs.orst.edu

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

Keywords: Update programming, type changes, type correctness, soundness of updates, Haskell, lambda calculus

1 Introduction

A major fraction of all programming activities is spent in the process of updating programs in response to changed requirements. The way in which these updates are performed has a considerable influence on the reliability, efficiency, and costs of this process. Text editors are a common tool used to change programs, and this fact causes many problems: for example, it happens quite often that, after having performed only a few minor changes to a correct program, the program consists of syntax and type errors. Even worse, logical errors can be introduced by program updates that perform changes inconsistently. These logical errors are especially dangerous because they might stay in a program undetected for a long time. These facts are not surprising because the "text-editor method" reveals a low-level view of programs, namely that of sequences of characters, and the operation on programs offered by text editors is basically just that of changing characters in the textual program representation.

Alternatively, one can view programs as abstract data types and program changes as well-defined operations on the program ADT. Together with a set of combinators, these basic update operations can then be used to write arbitrarily

* Technical Report TR02-60-09, Department of Computer Science, Oregon State University, October 2002

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

Generic updates can be collected in libraries that facilitate the reuse of updates and that can serve as a repository for executable software maintenance knowledge. In contrast, with the text-editor approach, each update must be performed on its own. At this point the safety of update programs shows an important advantage: whereas with the text-editor approach the same (or different) errors can be made over and over again, an update program satisfying the safety criteria will preserve the correctness for all object programs to which it applies. In other words, the correctness of an update is established once and for all. One simple, but frequently used update is the safe (that is, capture-free) renaming of variables. Other examples are extending a data type by a new constructor, changing the type of a constructor, or the generalization of functions. In all these cases the update of the definition of an object must be accompanied by according changes to all the uses of the object. Many more examples of generic program updates are given by program refactorings [16] or by all kinds of so-called "cross-cutting" concerns in the fast growing area of aspect-oriented programming [1, 13, 23, 5], which demonstrates the need for tools and languages to express program changes.

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

Our goal is *not* to replace the use of text editors for programming; rather, we would like to complement it: there will always be small or simple changes that can be most easily accomplished by using an editor. Moreover, programmers are used to writing programs with their favorite editor, so we cannot expect that they will instantly switch to a completely new way of performing program updates. The often described phenomenon of "resistance to change" makes this situation even less likely [4, 11]. However, there are occasions when a tedious task calls for automatic support. We can add safe update program for frequently used tasks to an editor, for instance, in an additional menu.[1]

Writing update programs, like meta programming, is in general a difficult task—probably more difficult than creating "normal" object programs. The proposed

[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.

approach does not imply or suggest that every programmer is supposed to *write* update programs. The idea is that update programs are written by a experts and used by a much wider audience of programmers (for example, through 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 how update programs.

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

2 Update Programming

To give an impression of the concept of update programming we show some updates to Haskell programs and how they can be implemented in HULA, the Haskell Update LAnguage [15] that we are currently developing. We also briefly indicate that the presented concepts apply more generally to other languages. We therefore also sketch a small Java example.

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

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

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

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

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

The update can be read as follows: the `con` update operation adds the type `Int` as a new first parameter to the definition of the `Node` constructor. The notation

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

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

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

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

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

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

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

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

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

library, so that constructor extensions as the one for `Node` can be expressed as applications of `extCon` [15]. For example, the size update can then be expressed by:

```
extCon Node Int succ 1
```

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

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

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. Then the function definition is extended by a new parameter and the expressions to be abstracted are replaced in the function body by this new variable. In addition, all applications of the function have 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 e = fun f {w} : {e/w} in 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} in
     fun eval : ‘case x ‘of {Con ‘c -> Con ‘c} cs
```

The extension of the `case` expression in `eval` expresses that constants evaluate to themselves. We can use the following update to extend the lambda calculus implementation by `let` expressions:

```
data Lam : cs { | Let String Lam Lam} in
     fun eval : ‘case x ‘of
          {Let ‘v ‘d ‘e -> ‘subst ‘d ‘v ‘e} cs
```

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.

Finally, we would like to mention that although we use Haskell in examples, the idea and concepts of update programming translate to other programming languages as well. For instance, the size example could be implemented in Java

as a class `Tree` with an `int` field `value` and two `Tree` fields `left` and `right` as well as a method `insert`. The update could be expressed by a `class` update that creates a new `int` field `size`, inserts an initialization into the constructor (at the beginning), and extends the `insert` method by a `size` incrementation (also at the beginning of the method):

```
class Tree:
  {int size}
  constr {this.size = 1;} c
  method insert: {size++;} m
```

Since Java is much more verbose than Haskell, we use underlining to mark meta variables.

3 Related Work

There is a large body of work on impact analysis that tries to address the problems that come with performing changes to software [2, 7]. However, we know of no work that attempts to exploit impact analysis to perform fully automated software changes.

Performing program updates in a more structured way is actually not a new idea. There exist a couple of program editors that can guarantee syntactic or even type correctness and other properties of changed programs. Examples for such systems are Centaur [10, 27], the synthesizer generator [26], or *CYNTHIA* [37, 36]. 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, 22], which directly relates proofs of type correctness with programs. In this respect it is very similar to proof editors like ALF [21], however, in contrast to ALF, proofs are not the main objective of *CYNTHIA* but rather used as a gluing representation between programs and their properties.

We have introduced a language-based view of program updates in [14]. 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. Viewing programs as abstract data types goes beyond the idea of syntax-directed program editors because it allows a programmer to combine basic updates into update programs that can be stored, reused, changed, shared, and so on. The program 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 the lambda calculus

updates indicate, 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. Again, take the lambda calculus updates as an example.

Programs that manipulate programs are also considered in the area of meta programming [29]. However, existing meta programming systems, such as MetaML [30], 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 [35]. Among these, only software rephrasing and refactoring work on one and the same language. Refactoring [16] is an area of fast growing interest with a few existing tools to perform refactoring automatically [28]. 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 is aspect-oriented programming [1], which is concerned with performing "cross-cutting" changes to a program. AspectJ [20, 3], Hyper/J [23, 19], and composition filters [5, 12] are some 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.

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 [24]. 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 [8, 9]. Visser has extended the set of strategy operators by generic term traversals [34], pattern matching operators [31], and other rewrite strategies that are specifically useful for language processing [32] and has put all these parts together into a system for program transformation, called *Stratego* [33]. These proposals allow a very flexible specification of rule application strategies, but they do not guarantee type correctness of the transformed programs.

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 [25].

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. It is principally possible to write update programs in such a two-level lambda calculus. However, symbols must retain their types over transformations whereas in our approach it is possible that symbols change their type (and name).

4 The Object Language

To keep the following description short and simple, we use lambda calculus together with a standard Hindley/Milner type system as the working object language. The syntax of lambda-calculus expressions is shown in Figure 1.

$$e \;::=\; c \mid v \mid e\ e \mid \lambda v.e \mid \texttt{let}\ v\,\texttt{=}\,e\ \texttt{in}\ e$$

Fig. 1. Abstract syntax of lambda calculus.

Constants are taken from a set C that is indexed by the available base types b. In addition to the base types, we have type variables (a) and function types; see Figure 2. 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 $\bar{a}$. 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.

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

Fig. 2. Types for lambda calculus.

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 are shown in Figure 3.

$$\text{CON}_{\vdash}\ \frac{c \in C_b}{\Gamma \vdash c : b} \qquad \text{VAR}_{\vdash}\ \frac{\Gamma(v) = \forall \bar{a}.t' \quad [t_i/a_i]t' = t}{\Gamma \vdash v : t}$$

$$\text{ABS}_{\vdash}\ \frac{\Gamma, v : t' \vdash e : t}{\Gamma \vdash \lambda v.e : t' \to t} \qquad \text{APP}_{\vdash}\ \frac{\Gamma \vdash e : t' \to t \quad \Gamma \vdash e' : t'}{\Gamma \vdash e\ e' : t}$$

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

Fig. 3. 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.

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

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

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 will comment on this in Section 5.2.

5 The Update Calculus

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

5.1 Rules

A rewrite rule has the form:

$$l \rightsquigarrow r$$

where l and r are expressions that might contain meta variables, that is, variables that are different from object variables and can represent arbitrary expressions. Expressions that possibly contain meta variables are called *patterns*; their syntax is defined in Figure 4.

$$p \ ::= \ \underline{m} \mid c \mid v \mid p\ p'$$

Fig. 4. Patterns.

If we remove meta variables, 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.

We extend the type system from Figure 3 by the following rule for meta variables:

$$\text{META}_{\vdash}\ \frac{\Gamma(\underline{m}) = t}{\Gamma \vdash \underline{m} : t}$$

This rule is similar to the rule $\text{VAR}_{\vdash}$, 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 meta variables in l. The fact that a pattern like l matches an expression e (under the substitution σ) is also written as: $l \succ e$ ($l \underset{\sigma}{\succ} e$). We assume that l is linear, that is, l does not contain any meta variable twice. The result of the update operation is $\sigma(r)$, that is, r with all meta variables being substituted according to σ. If l does not match e, the update described by the rule is not performed, and e remains unchanged.

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

5.2 Update Combinators

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

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 $\{v \rightsquigarrow v' \colon u_d\} u_u$ for an update that renames v to v', changes v's definition by u_d, and all of its uses by u_u. (We also call u_d the *definition update* and u_u the *use update*.) Note that u_u is always applied recursively, whereas u_d is only applied to the root of the definition. However, to account for recursive `let` definitions we apply u_u also recursively to the result obtained by the update u_d.

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 can always identify precisely all the parts of an update where fresh variables are required so that we can integrate the generation of fresh variables into the semantics for updates.

The preceding discussion leads to the syntax of updates shown in Figure 5.

We use x to range over variables (v) and meta variables ($\underline{m}$), which means that we can use a scope update to update specific bindings (by using an object variable) or to apply to arbitrary bindings (by using a meta variable). Either one

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

Fig. 5. Syntax of updates.

of the variables (but not both) can be missing. These special cases describe the creation or removal of a binding. In both cases, we have an expression instead of a definition update. This expression is required in the case of binding removal where it is used to replace all occurrences of the removed variable. (Note that e must neither contain x nor a possible object variable that matches x in case x is a meta variable.) In the case of binding creation, the expression is optional and is used, if present, as a definition for the newly introduced variable, which must be an object variable. We use an abbreviated notation for scope updates that do not change names, that is, we write $\{v\colon u_d\}u_u$ instead of $\{v \rightsquigarrow v\colon u_d\}u_u$. The updates of either the defining expression or the scope can be empty, which means that there is no update for that part. The updates are then simply written as $\{v\colon u_d\}$ and $\{v\}u_u$, respectively, and are equivalent to updates $\{v\colon u_d\}\iota$ and $\{v\colon \iota\}u_u$, respectively.

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

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 introduces a new binding for `w` and replaces all occurrences of a particular constant expression (here `1`) by `w` in the definition of `f`. u's use update makes sure that all uses of `f` are extended by supplying a new argument for the newly introduced parameter. Here we use the same expression that was generalized in `f`'s definition, which preserves the semantics of the program. (Note that in order to express a function like `genFun` we have to extend the update calculus by abstraction, application, and variables.)

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

performing an update. A `case` update can then be translated into an alternative of change-scope updates. For example, the translation of the size update yields:

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

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

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

5.3 Semantics of Updates

In the definition of the semantics for alternative updates $(u\,;\,u')$ we need a judgment of the form $u \succcurlyeq_@ e$ that tells whether or not the update u applies to (the root of) expression e: we apply u' only to e if $u \succcurlyeq_@ e$ does *not* hold. (Note that we cannot simply use a test like $[\![u]\!]_\rho(e) \neq e$ because, for example, ι applies to any expression but does not change it.) The formalization of this relationship is straightforward: for identity, rules, and recursion we have: $\iota \succcurlyeq_@ e$, $l \rightsquigarrow r \succcurlyeq_@ e$ if $l \succ e$, and $\downarrow u \succcurlyeq_@ e$ if $u \succcurlyeq_@ e$, respectively. An alternative applies if either sub-update applies, that is, $u\,;\,u' \succcurlyeq_@ e$ if $u \succcurlyeq_@ e$ or $u' \succcurlyeq_@ e$. Any scope update of the form $\{x \rightsquigarrow \ldots\} \ldots$ principally applies to any expression that binds a variable v with $x \succ v$, that is, any expression of the form `let` v `=` d `in` e, $\lambda v.e$, or $(\lambda v.e)\ e'$. More precisely, a scope deletion does not apply to β-redexes, and a scope change applies to a lambda abstraction only if the definition update u_d is ι. Finally, the insert scope update $\{\rightsquigarrow v \ldots\} \ldots$ applies to any expression.

It might seem that the definition of $\downarrow u \succcurlyeq_@ e$ is overly simplistic because u might apply by virtue of recursion somewhere down deep in e. However, for our specific purpose, namely deciding whether or not to apply the second update in an alternative, the chosen definition is sufficient since we do not have to deal with updates like $\downarrow u\,;\,u'$ because we use $\downarrow$ only implicitly in use updates.

An update u is applied to an expression e relative to a set of names that define the scope of the update. These scopes are constructed by scope updates and are reduced when recursion moves into expressions that bind variables already contained in the scope. A rule can be applied only if all the free (object) variables of the left side are contained in the scope. We also have to consider the set of variables that are bound by the expression being currently 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 be bound in the expression being updated. We can represent both sets 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 update expression but that are not in scope. We need two operations for moving variables between ρ_S and ρ_B:

$$\begin{array}{rcl}(\rho_S, \rho_B)\langle v & := & (\rho_S \cup \{v\}, \rho_B - \{v\}) \\ (\rho_S, \rho_B)\rangle v & := & (\rho_S - \{v\}, \rho_B \cup \{v\})\end{array}$$

We define the semantics as judgments of the form $[\![u]\!]_\rho(e) = e'$; see Figure 6. The semantics of the recursion operator is standard; we have therefore omitted it to save space. We also have omitted all no-change rules and we make by default a "no-change assumption", that is, when none of the shown rules applies, $[\![u]\!]_\rho(e)$ yields e. We also write $[\![u]\!](e)$ instead of $[\![u]\!]_{(\varnothing,\varnothing)}(e)$.

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 is neither bound in e or the current context (ρ_B). (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.) If v has this property, it will be chosen, otherwise an appropriate name will be constructed (for example, by repeatedly appending a prime symbol until a fresh symbol is found). In the semantics rules shown in Figure 6 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. The predicate is defined as:

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

We also use the notations $u^\langle$ for the update u with all free occurrences of x substituted by v, $u^\rangle$ for the update u with all free occurrences of x' substituted by w, and $u^{\langle\rangle}$ for the update u with all free occurrences of x substituted by v and with all free occurrences of x' substituted by w.

Pairs of updates can be classified according to their possible interaction. Two updates u and u' are called *competing* iff $\exists e : u \succ_@ e \wedge u' \succ_@ e$. Dually, u and u' are called *independent*, written as $u \| u'$, iff they are not competing. The goal of these definitions is to obtain criteria, for example, for the well-definedness of updates and for their type safety. In particular, we require type changes for competing updates to be compatible in a way to be defined below, whereas type changes for independent updates will result in alternative type changes.

6 A Type System for Updates

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

6.1 Type Changes

Since updates denote changes of expressions that may involve a change of their types, the types of updates are described by *type changes*. Type changes are essentially represented by pairs of types. Changes that apply (through recursion) in

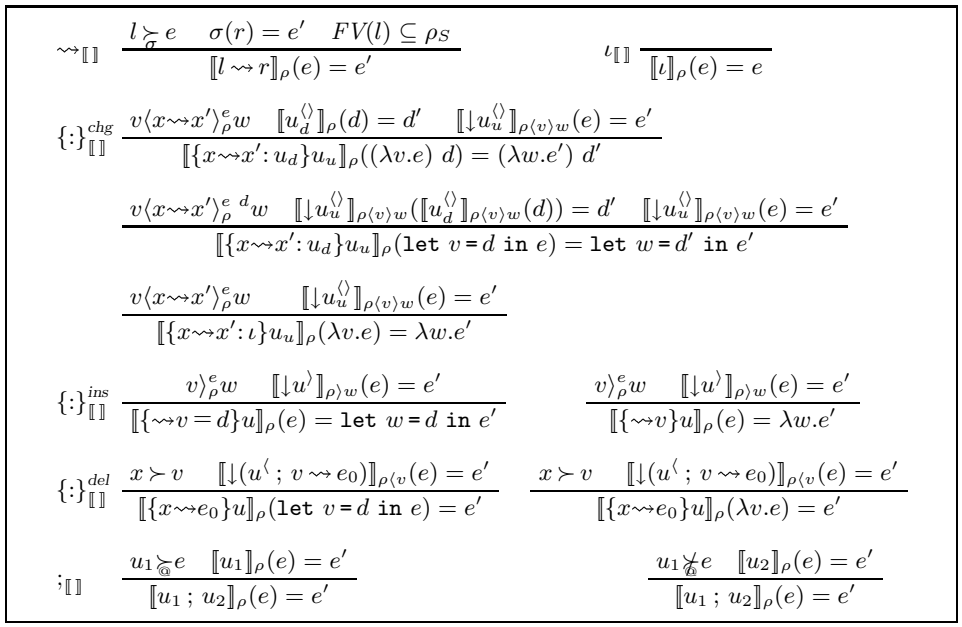

Fig. 6. Semantics of updates.

subexpressions are described by context type changes, which are given pairs of context types. Context types can be thought of as type derivations in which a specific subpart (a type) is exposed. A context type is given by an application of a type context to a type. A *type context* is either empty (ϵ) or a pair $t' \mapsto t$ where t' is the exposed type of the derivation for the type t (t' can also be empty). The application of a type context C to a type t yields a *context type* and is written as $C\langle t \rangle$. The meaning of a context type is given by the following equations.

$$\epsilon\langle t \rangle = t$$
$$t_1 \mapsto t_2\langle t \rangle = \begin{cases} t_2 & \text{if } t \succ t_1 \vee t_1 \text{ is empty} \\ \text{error} & \text{otherwise} \end{cases}$$

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 describe the type changes for use updates in scope updates. The type changes might refer to the types of the variables manipulated by the scope update, whose types can be constrained by the context of the object language expressions in which they matched during an update. To describe this dependency we also introduce the notion of a *constrained type*, which is a type t together with a type context C that possibly constrains t, written as $t_{|C}$. The meaning is:

$$t_{|\epsilon} = t$$
$$t_{|t_1 \mapsto t_2} = \begin{cases} t & \text{if } t_1 \text{ is empty} \vee t \neq a \\ t_1 & \text{otherwise} \end{cases}$$

To summarize, a context type evaluates for non-empty contexts to the root/top-level type of the applied context, which means to ignore the inner/nested type,

whereas a constrained type evaluates to the exposed type of the derivation (if it is not empty). An empty context has no effect on the type argument of a context type or constrained type. The syntax of contexts and type changes is summarized in Figure 7.

$$\begin{array}{rcl} \delta & ::= & \tau \rightsquigarrow \tau \quad | \quad \delta|\delta \\ \tau & ::= & b \mid a \mid \tau \to \tau \mid C\langle\tau\rangle \mid \tau_{|C} \\ C & ::= & \epsilon \mid [t]{\mapsto}t \end{array}$$

Fig. 7. Type changes.

Let us consider some examples. The update $u = \{\rightsquigarrow\mathtt{w}\}\mathtt{1} \rightsquigarrow \mathtt{w}$ changes the expression `1` to λ`w.w` with the type change `Int` $\rightsquigarrow$ `a->a`. However, applied to `odd 1`, u yields λ`w.odd w` with the type change `Bool` $\rightsquigarrow$ `Int->Bool`. How can we describe and explain these two different type changes? First, `w` is a new variable so we assume a fresh type variable `a` for it. Second, the type of the use update `1` $\rightsquigarrow$ `w` is determined as `Int` $\rightsquigarrow$ `a`. However, since the use update is performed recursively, the type change for the new-scope update is described using context types, that is, instead of `Int` $\rightsquigarrow$ `a` we expect something like $C\langle$`Int`$\rangle \rightsquigarrow C\langle$`a`$\rangle$. Third, the type for the newly introduced abstraction has to be taken into account. Here we have to observe that the type of `w` cannot be `a` in general, because `w` might be (again through the recursive application of the rule) placed into an expression context that constrains `w`'s type. This can be expressed by using the constrained type $\mathtt{a}_{|C}$ for `w`. Therefore, the type change of u is $C\langle\mathtt{Int}\rangle \rightsquigarrow \mathtt{a}_{|C}\mathtt{->}C\langle\mathtt{a}_{|C}\rangle$, which can be instantiated with the two contexts ϵ and `Int`$\mapsto$`Bool`, respectively, to the above shown type changes. As another example we consider the renaming update $u = \{\mathtt{x} \rightsquigarrow \mathtt{y}\}\mathtt{x} \rightsquigarrow \mathtt{y}$. 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 $C\langle\mathtt{a}_{|C}\rangle \rightsquigarrow C\langle\mathtt{a}_{|C}\rangle$. The context and constraint C results for the same reason as in the previous example. Applying u to the expression λ`x.1` yields λ`y.1` with a type change `a->Int` $\rightsquigarrow$ `a->Int`, which can be obtained from u's type change by using the corresponding context $\mapsto$`a->Int`. Similarly, u changes λ`x.odd x` to λ`y.odd y` with a type change `Int->Bool` $\rightsquigarrow$ `Int->Bool`. This type change is u's type change specialized for the context `Int`$\mapsto$`Int->Bool`.

Since contexts can be nested, we define their composition as follows.

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

It is easy to verify that with this definition we have

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

which means that the outermost contexts affect context types, whereas the innermost context is relevant for constrained types. Finally, since the inference rules generate, in general, context constraints for arbitrary type changes, we have to explain how contexts are propagated through type changes to types:

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

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 $t \sqsupseteq t' \to t$. The rationale for this definition is that two updates u and u' of different types $t_1 \rightsquigarrow t_2$ and $t'_1 \rightsquigarrow t'_2$, respectively, can be considered well typed in an alternative $u\,;\,u'$ if one type change is an applicative instance of the other, that is, if $t_1 \rightsquigarrow t_2 \sqsupseteq t'_1 \rightsquigarrow t'_2$ or $t'_1 \rightsquigarrow t'_2 \sqsupseteq t_1 \rightsquigarrow t_2$, because in that case one update is just more specific than the other. For example, in the update

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

the first rule of the alternative `f x` $\rightsquigarrow$ `f x 1` has the type change `Int` $\rightsquigarrow$ `Int` whereas the second rule `f` $\rightsquigarrow$ `f 1` has the type change `Int->Int`$\rightsquigarrow$`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` $\rightsquigarrow$ `Int` being an applicative instance of `Int->Int` $\rightsquigarrow$ `Int->Int`. The relationship is defined in Figure 8. The definition for alternative type changes can identify applicative instance pairs in alternatives, which can then be generalized (see below).

$$\textsc{refl}_{\sqsupseteq}\ \frac{}{\tau \sqsupseteq \tau} \qquad c_{\sqsupseteq}\ \frac{\tau \sqsupseteq \tau'}{C\langle\tau\rangle \sqsupseteq C\langle\tau'\rangle} \qquad \frac{\tau \sqsupseteq \tau'}{\tau_{|C} \sqsupseteq \tau'_{|C}}$$

$$\to_{\sqsupseteq}\ \frac{\tau \sqsupseteq \tau_2}{\tau \sqsupseteq \tau_1 \to \tau_2} \qquad \rightsquigarrow_{\sqsupseteq}\ \frac{\tau_1 \sqsupseteq \tau'_1 \quad \tau_2 \sqsupseteq \tau'_2}{\tau_1 \rightsquigarrow \tau_2 \sqsupseteq \tau'_1 \rightsquigarrow \tau'_2}$$

$$|_{\sqsupseteq}\ \frac{\tau_1 \rightsquigarrow \tau_2 \sqsupseteq \tau'_1 \rightsquigarrow \tau'_2}{\tau_1 \rightsquigarrow \tau_2 | \delta \sqsupseteq \tau'_1 \rightsquigarrow \tau'_2 | \delta'}$$

Fig. 8. Applicative instance.

We say that two type changes δ and δ' are *applicative-instance compatible*, written as $\delta \equiv \delta'$, if $\delta \sqsupseteq \delta'$ or $\delta' \sqsupseteq \delta$. Given two applicative-instance-compatible types we can compute their generalization as follows.

$$gen(\tau_1 \rightsquigarrow \tau_2, \tau'_1 \rightsquigarrow \tau'_2) = \begin{cases} \tau_1 \rightsquigarrow \tau_2 & \text{if } \tau_1 \rightsquigarrow \tau_2 \sqsupseteq \tau'_1 \rightsquigarrow \tau'_2 \\ \tau'_1 \rightsquigarrow \tau'_2 & \text{if } \tau'_1 \rightsquigarrow \tau'_2 \sqsupseteq \tau_1 \rightsquigarrow \tau_2 \end{cases}$$

$$gen(\delta_1|\delta, \delta_2|\delta') = \begin{cases} gen(\delta, gen(\delta_1, \delta_2)|\delta') & \text{if } \delta_1 \equiv \delta_2 \\ \delta_1|\delta_2|gen(\delta, \delta') & \text{otherwise} \end{cases}$$

Since u and u' have to be independent to generate a type-change alternative (see Figure 9), it follows that $|$ is commutative. Therefore, alternative type changes can be treated like bags, that is, we can reorder contained type changes as needed in the application of *gen*.

With this generalization operation, the type system infers a most general type change for alternative updates whose types are applicative-instance compatible. On the other hand, the types of independent update alternatives is collected in type-change alternatives. For example, the updates $u =$ `1` $\rightsquigarrow$ `2` and $u' =$ `True` $\rightsquigarrow$ `False` are independent, which allows them to be placed in an alternative update because their incompatible type changes result in a type-change alternative, that is, $u\,;\,u' ::$ `Int`$\rightsquigarrow$`Int`|`Bool`$\rightsquigarrow$`Bool`.

Finally, note that a type change $t \rightsquigarrow t'$ does not necessarily mean that an update $u : t \rightsquigarrow t'$ maps an expression e of type t to an expression of type t', because u might

not apply to e and thus we might get $[\![u]\!](e) = e$ of type t. Thus, the information about an update causing some type change is always to be interpreted as "optional" or "contingent on the applicability of the update".

6.2 Type Change Inference

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

$$;_{\triangleright}\ \frac{\Delta \triangleright u :: \delta \qquad \Delta \triangleright u' :: \delta' \qquad \delta \equiv \delta'}{\Delta \triangleright u\,;\,u' :: gen(\delta, \delta')} \qquad \frac{\Delta \triangleright u :: \delta \qquad \Delta \triangleright u' :: \delta' \qquad u \| u'}{\Delta \triangleright u\,;\,u' :: \delta | \delta'}$$

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

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

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

Fig. 9. Type change system.

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

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

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

$$\Delta_\ell := \{x : t \mid x{\rightsquigarrow}x' :: t{\rightsquigarrow}t' \in \Delta\} \cup \{x : t \mid x :_\ell t \in \Delta\}$$
$$\Delta_r := \{x' : t' \mid x{\rightsquigarrow}x' :: t{\rightsquigarrow}t' \in \Delta\} \cup \{x' : t' \mid x' :_r t' \in \Delta\}$$

The type-change rules are defined in Figure 9. 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 $\tau \overrightarrow{\ }_\ell \delta$ ($\tau \overrightarrow{\ }_r \delta$) to extend the argument (result) type of a type change to a function type. The definition is as follows.

$$\tau \overrightarrow{\ }_\ell (\tau_l \rightsquigarrow \tau_r) := (\tau \to \tau_l) \rightsquigarrow \tau_r$$
$$\tau \overrightarrow{\ }_r (\tau_l \rightsquigarrow \tau_r) := \tau_l \rightsquigarrow (\tau \to \tau_r)$$
$$\tau \overrightarrow{\ }_\ell (\delta | \delta') := (\tau \overrightarrow{\ }_\ell \delta) | (\tau \overrightarrow{\ }_\ell \delta')$$
$$\tau \overrightarrow{\ }_r (\delta | \delta') := (\tau \overrightarrow{\ }_r \delta) | (\tau \overrightarrow{\ }_r \delta')$$

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

6.3 Soundness of the Update Type System

In this section we present the main result of this paper: a characterization of a class of updates by a normal form so that the application of a well-typed update from this class is guaranteed to preserve the well-typing of any transformed object language expressions. An update that, when applied to a well-typed expression, yields again a well-typed expression is called *safe*. In other words, we will show that typeable updates in normal form are safe. The normal form 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). This rule prevents ill-typed applications of changed symbols as well as unbound variables.

(B) No use update can introduce a non-generalizing type change, that is, for each use update that has a type change $t \rightsquigarrow t' | \delta$ we require: $t' \succ t$. This rule prevents that changed symbols break the well-typing of their contexts.

An intuitive explanation of why such a normal form implies safety for well-typed updates can be obtained by looking at all 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 normal-form constraints. We can find out about possible type errors by looking at the type system for lambda calculus (see Figure 3): Essentially, type inference fails either in the rule $\textsc{var}_{\vdash}$ if the type for a variable cannot be found in the type environment or in the rule $\textsc{app}_{\vdash}$ 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 $\textsc{abs}_{\vdash}$ and $\textsc{let}_{\vdash}$ eventually refer to $\textsc{var}_{\vdash}$ and $\textsc{app}_{\vdash}$ 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 $\textsc{var}_{\vdash}$ rule, or they might be bound by an enclosing λ or `let` and thus might change the type that is obtained by the $\textsc{var}_{\vdash}$ rule, thus breaking eventually the $\textsc{app}_{\vdash}$ rule. These kind 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 sub-expression; otherwise, a rule cannot change only one part of an application. However, since application to sub-expressions 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 t to t', 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 $u = \texttt{v} \rightsquigarrow \texttt{e}$ causes all occurrences to be changed (not necessarily by this rule, but no occurrence of `v` is left unchanged), and (ii) the type change $t \rightsquigarrow t_2$ derived by the type system is required to be "generalizing" (that is, $t_2 \succ t$, see below), which ensures that `e` fits all contexts typewise.

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

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

An update u is in *normal form* 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 normal-form update u to a well-typed expression e, the following two cases can occur: (1) u does not apply to e ($u \not\succeq_{@} e$). In this case e is not changed by u and remains well typed. (2) u applies to e and changes it into e'. In that case we have to show that from the result type of u we can infer the type of e'.

The following lemma captures the obvious fact that an update that does not apply to an expression leaves the type of that expression unchanged (because the update does not change the expression):

Lemma 1. $\quad u \not\succeq_{@} e \wedge \Gamma \vdash e : t \implies \Gamma \vdash [\![u]\!](e) : t$

The next lemma gives the main result, namely that normal-form updates preserve the type correctness of expressions to which they apply.

Lemma 2 (NF Soundness). *If u is in normal form, then*

$$\Delta \triangleright u :: [\tau_{\overrightarrow{X}}] C\langle t \rangle \rightsquigarrow C\langle t' \rangle | \delta \wedge u \succeq_{@} e \wedge \Delta_\ell \vdash e : t \implies \Delta_r \vdash [\![u]\!]_{(dom(\Delta_\ell), \varnothing)}(e) : C\langle t' \rangle$$

Finally, the following theorem is a corollary that combines the results from Lemmas 1 and 2.

Theorem 1. *If u is in normal form, then*

$$\Delta \triangleright u :: [\tau_{\overrightarrow{X}}]C\langle t\rangle \rightsquigarrow C\langle t'\rangle | \delta \wedge \Delta_\ell \vdash e : t \implies \Delta_r \vdash [\![u]\!]_{(dom(\Delta_\ell),\varnothing)}(e) : t'' \wedge (t'' = t \vee t'' = C\langle t'\rangle)$$

Let us consider the safety of some of the presented example updates. The function generalization update is safe, which can be checked by applying the definitions of normal form and the rules of the type-change system. The first size update 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`. The lambda-calculus updates are safe; however, the updates might leave some functions that use expressions of the `Lam` data type unexpanded. Now, for programs that only use `Node` expressions in a function `insert` the second size update *is* safe. Similarly, programs that use `Lam` expressions only in a function `eval`, the lambda calculus updates do not introduce non-exhaustive `case` expressions. We discuss this aspect briefly in the next section when we talk about extensions of the system.

7 Conclusions and Future Work

We have introduced an update calculus for expressing changes to lambda-calculus expressions. The type-change system for the update calculus ensures that updates in normal form are safe, that is, they preserves the well typing of lambda-calculus expressions. We have indicated how the presented calculus can serve as a basis for high-level update languages that can be translated into the calculus. We already have an implementation for the update calculus. Currently, we are working on the implementation for the update language for Haskell.

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

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 normal form that is required to guarantee safety of updates is 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.

Allow non-generalizing type changes. Currently, updates in normal form 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 captured by requiring an update that applies to all possible contexts and basically inserts expressions that revert the type change. However, with the concept of conditional update safety we can relax the covering criterion.

References

1. ACM. *Communications of the ACM*, volume 44(10), October 2001.
2. R. S. Arnold and S. A. Bohner. Impact Analysis – Towards a Framework for Comparison. *IEEE Int. Conf. on Software Maintenance*, pp. 292–301, 1993.
3. AspectJ, 2002. `http://aspectJ.org`.
4. M. W. Bauer, editor. *Resistance to New Technology: Nuclear Power, Information Technology, and Biotechnology*. Cambridge University Press, Cambridge, NY, 1997.
5. L. Bergmans and M. Askit. Composing Crosscutting Concerns using Composition Filters. *Communications of the ACM*, 44(10):51–57, 2001.
6. N. Bjørner. Type Checking Meta Programs. *Workshop on Logical Frameworks and Meta-Languages*, 1999.
7. S. A. Bohner and R. S. Arnold, editors. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, 1996.
8. B. Borovanský, C. Kirchner, H. Kirchner, P. E. Moreau, and C. Ringeissen. Rewriting with Strategies in ELAN: a Functional Semantics. *Int. Journal of Foundations of Computer Science*, 2001. To appear.
9. B. Borovanský, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN: A Logical Framework Based on Computational Systems. *Workshop on Rewriting Logic and Applications*, 1996.
10. P. Borras, D. Clèment, T. Despereaux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. *3rd ACM SIGSOFT Symp. on Software Development Environments*, pp. 14–24, 1988.
11. B. M. Bouldin, editor. *Agents of Change: Managing the Introduction of Automated Tools*. Yourdon Press, Englewood Cliffs, NJ, 1989.
12. Composition Filters, 2001. `http://trese.cs.utwente.nl/composition_filters`.
13. T. Elrad, M. Askit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing Aspects of AOP. *Communications of the ACM*, 44(10):33–39, 2001.
14. M. Erwig. Programs are Abstract Data Types. *16th IEEE Int. Conf. on Automated Software Engineering*, pp. 400–403, 2001.
15. M. Erwig and D. Ren. A Rule-Based Language for Programming Software Updates. *3rd ACM SIGPLAN Workshop on Rule-Based Programming*, pp. 67–77, 2002.
16. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 1999.
17. S. Horwitz, J. Prins, and T. Reps. Integrating Non-Interfering Versions of Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, 1989.
18. W. A. Howard. The Formulae-As-Types Notion of Construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479–490. Academic Press, 1980.
19. Hyper/J, 2001. `http://www.alphaworks.ibm.com/tech/hyperj`.
20. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.
21. L. Magnusson and B. Nordström. The ALF Proof Editor and its Proof Engine. *Types for Proofs and Programs*, LNCS 806, pp. 213–237, 1994.
22. P. Martin-Löf. Constructive Mathematics and Computer Programming. *6th Int. Congress for Logic, Methodology and Philosophy of Science*, pp. 153–175, 1979.
23. H. Ossher and P. Tarr. Using Multidimensional Separation of Concerns to (Re)shape Evolving Software. *Communications of the ACM*, 44(10):43–50, 2001.

24. F. Pfenning. Logical Frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 21. Elsevier Science Publishers, Amsterdam, NL, 2001.
25. T. Reps. Algebraic Properties of Program Integration. *Science of Computer Programming*, 17:139–215, 1991.
26. T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989.
27. L. Rideau and L. Thèry. An Interactive Programming Environment for ML. Rapport de Recherche 3139, INRIA, Sophia Antipolis, 1997.
28. D. Roberts and J. Brant. Refactoring Tools. In M. Fowler, editor, *Refactoring: Improving the Design of Existing Code*, chapter 14, pp. 309–352. Addison-Wesley, Reading, MA, 1999.
29. T. Sheard. Accomplishments and Research Challenges in Meta-Programming. *2nd Int. Workshop on Semantics, Applications, and Implementation of Program Generation*, LNCS 2196, pp. 2–44, 2001.
30. W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.
31. E. Visser. Strategic Pattern Matching. *10th Int. Conf. on Rewriting Techniques and Applications*, LNCS 1631, pp. 30–44, 1999.
32. E. Visser. Language Independent Traversals for Program Transformation. *Workshop on Generic Programming*, 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.
33. E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. *12th Int. Conf. on Rewriting Techniques and Applications*, LNCS 2051, 2001.
34. E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. *3rd ACM Int. Conf. on Functional Programming*, pp. 13–26, 1998.
35. E. Visser, et al. The Online Survey of Program Transformation. `http://www.program-transformation.org/survey.html`.
36. J. Whittle, A. Bundy, R. Boulton, and H. Lowe. An ML Editor Based on Proofs-as-Programs. *9th Int. Symp. on Programming Language Implementation and Logic Programming*, LNCS 1292, pp. 389–405, 1997.
37. J. Whittle, A. Bundy, and H. Lowe. An Editor for Helping Novices to Learn Standard ML. *14th Int. Conf. on Automated Software Engineering*, 1999.