A Rule-Based Language for Programming Software Updates

Martin Erwig
Department of Computer Science
Oregon State University
erwig@cs.orst.edu

Deling Ren
Department of Computer Science
Oregon State University
rende@cs.orst.edu

Abstract

We describe the design of a rule-based language for expressing changes to Haskell programs in a systematic and reliable way. The update language essentially offers update commands for all constructs of the object language (a subset of Haskell). The update language can be translated into a core calculus consisting of a small set of basic updates and update combinators. The key construct of the core calculus is a scope update mechanism that allows (and enforces) update specifications for the definition of a symbol together with all of its uses.

The type of an update program is given by the possible type changes it can cause for an object programs. We have developed a type-change inference system to automatically infer type changes for updates. Updates for which a type change can be successfully inferred and that satisfy an additional structural condition can be shown to preserve type correctness of object programs.

In this paper we define the Haskell Update Language HULA and give a translation into the core update calculus. We illustrate HULA and its translation into the core calculus by several examples.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Functional Programming; D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.7 [**Software Engineering**]: Maintenance; F.3.1 [**Logics and Meanings of Programs**]: Reasoning about Programs

General Terms

Design, Languages, Reliability

Keywords

Update program, type change, type correctness, update safety, Haskell

3rd ACM SIGPLAN Workshop on Rule-Based Programming (RULE'02), Pittsburgh, PA, 2002

1 Introduction

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 a text editor, 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. Logical errors are especially dangerous because they can remain undetected for a long period. At the same time, these and other errors can cause a correct and perfectly running program to become instantly unusable. It is not surprising that this situation exists. The "text editor method" reveals a low-level view of programs that fails to reflect program structure, relying instead on sequences of single characters. The operation on programs offered by text editors is basically just that of changing characters in the textual program representation. Determining the consequences of resulting changes as they affect software quality is further complicated by a lack of tools.

We suggest viewing programs as abstract data types (ADTs) and performing program changes by applying well-defined ADT operations on the program. Employing these basic update ADT operations, arbitrarily complex *update programs* can be written that guarantee a high level of correctness for the resulting program. In particular, update ADT operations can prevent—in addition to syntax and type errors—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. Moreover, these tools cannot guarantee the different forms of correctness, since they have no knowledge of the languages' definitions, such as their grammar and scoping rules.

Update programs can be generic, that is, they can be applied to different programs. Thus, they can be collected in libraries that facilitate the reuse of updates and serve as repositories 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. While the same or different errors can be made again and again with the text editor approach, an update program satisfying certain safety criteria will preserve the correctness of all object programs to which it applies. In other words, the correctness of an update is established once and for all. Program update operations can also be integrated into program editors to offer safe high-level updates in an interactive way.

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, two distinct 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 reflects the changes performed much better than a partially changed object program that only shows the result of having applied a number of update steps. Second, independent updates can be defined and applied independently. For example, assume an update u_1 followed by an update u_2 (that does not depend on or interfere with u_1) is applied to a program. With the editor approach, we can undo u_2 and also u_2 and u_1, but we cannot undo just u_1. 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.

Our goal is not to replace text editors; many small or unstructured updates can be effectively performed by just using an editor. We rather want to extend the options programmers and software developers have in performing updates and structuring versions of their software. In particular, we are planning to extend text editors like Emacs or Vim by menus that offer interactive access to generic update functions to provide a simple access to reliable program update operations.

In this paper we describe an update language for the programming language Haskell, but the presented concepts can be also applied to other (programming) languages in similar ways. In the next section we illustrate the idea of update programming by a small example. Then after reviewing related work in Section 3, we discuss the design issues of the update language in Section 4. The design decisions lead to the definition of a small core calculus for expressing program updates, which is presented in in Section 5. In this section we also comment briefly on the formal semantics and type system of the core calculus and the type-safety result that is for lack of space presented in a separate paper. In Section 6 we define the syntax of the Haskell Update Language (HULA), and give a translation into the core update calculus. Conclusions given in Section 7 complete this paper.

2 Program Update Programs

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 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 these coordinated changes in our update language 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 specification of how to change all pattern matching rules that use the `Node` constructor follows after the keyword `in`: `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). `Leaf` patterns are left unchanged, and occurrences of the `Node` constructor within their corresponding return expressions are extended by `1`. As an alternative to the update to case expressions, the rule `Node {1}` extends all other `Node` expressions by `1`.

The shown update is safe in the sense that the produced object program will be syntax and type correct. Notice that this property does not only hold for the above example program, but for *any* type-correct input program! An intuitive reason is that the changed type of the `Node` constructor is accompanied by a corresponding change of all uses of `Node` (in patterns and expressions). Type safety can be ensured by a type system for the update language and structural constraints on update programs; see Section 5. The application of the update to the original program yields the following 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)
```

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 and the update for extending other `Node` expressions by `1` 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 `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. Although the shown update is safe for the example program, it is not safe in general since the extension of `Node` by the additional argument `1` is specified only as part of the `fun` update for `insert`; uses of `Node` outside of the `insert` function will not be changed. However, general type safety can be recovered by adding as an alternative to the `fun` update an update for all other `Node` patterns and `Node` expressions.

The backquote indicates that `insert` is a Haskell variable and not a variable of HULA. Haskell variables are also called *object variables* since they are variables of the object language to be transformed. In contrast, HULA variables are called *meta variables* since these are variables that are used to bind Haskell objects; in particular, meta variables will never appear in a Haskell program. In case of doubt, variables will be regarded as meta variables, so we need a syntactic means to prevent Haskell variables from being interpreted as meta variables. We are using the backquote symbol for this purpose. However, since meta variables can reasonably occur only at specific locations in an update program, we might omit the backquote when it is clear from the context that a variable is a Haskell variable. In particular, a meta variable has to be introduced always on the left-hand side of a rule (otherwise it would be unbound). Therefore, we can assume that unquoted variables that occur the first time on the right-hand side of a rule are meant to be object variables. The notions of "left", "right", and "rule" are related to our notation in the following way: $a\{l/r\}b$ is an abbreviation for the rule $a\,l\,b \rightsquigarrow a\,r\,b$ where the left part "l/" is optional, that is, $a\{r\}b$ is an abbreviation for $a\,b \rightsquigarrow a\,r\,b$ and thus means to insert r into the context $a\,b$. In the above example, `{Int} t` is an abbreviation for the rule `t` $\rightsquigarrow$ `Int t`, and `Node {succ s}` is an abbreviation for the rule `Node` $\rightsquigarrow$ `Node (succ s)`. With the described syntactic convention we can discern that in the first example `t` is a meta variable since it is introduced on the left hand side of the rule, whereas in the second example `succ` as well as `s` are object variables even though they are not marked by a quote, because they are not introduced on the left hand side of a rule and would thus be undefined when regarded as meta variables. By the same rule we can identify the variables `x` and `y` as meta variables. We also see the need for using the backquote for `insert`: without the backquote `insert` would be a meta variable and would match *all* function names in the Haskell program and not just the function `insert`. Note that we do not need a quoting mechanism for constructors like `Node` or `Leaf` since we (currently) do not have constructors in the update language. Therefore, constructors are always treated as object language constructors.

3 Related Work

In [7] we have proposed a language-based view of program updates. That paper describes a general model of programs, updates, and the preservation of arbitrary properties. It also discusses a way of ensuring type correctness for the simply-typed lambda calculus that is based on computing required and provided changes in type assumptions. 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 [5], the synthesizer generator [18], or *CYNTHIA* [28]. The view underlying these tools are either that of syntax trees or, in the case of *CYNTHIA*, proofs in a logical system for type information. Viewing programs as abstract data types 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 process of manipulating programs by other programs is usually called *meta-programming* [20]. However, existing meta-programming systems, such as MetaML [21], 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 [27]. Among these, only software rephrasing and refactoring [10] work on one and the same language. Refactoring is an area of fast growing interest with a few existing tools to perform refactoring automatically [19]. 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. For instance, AspectJ [11] is a tool that can be used to deal with aspects in Java programs. In [22] it is argued that modifications for legacy software should be automated. Moreover, processes and tools are described that can be employed for this purpose.

Our approach is based in part on applying update rules to specific parts of a program. There has been considerable work in the area of term rewriting to address this issue. Traditionally, rewrite systems have considered 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 [16]. 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 [3, 4]. Eelco Visser has extended the set of strategy operators by generic term traversals [26], pattern matching operators [23], and other rewrite strategies that are specifically useful for language processing [24] and has put all these parts together into a system for program transformation, called *Stratego* [25]. These proposals allow a very flexible specification of rule application strategies. Ralf Lämmel and Joost Visser have developed a series of transformation systems for Haskell (Tabaluga, Strafunski) that are based on the idea of generalized fold operations [12]. TXL [6] is a rule-based language for describing program transformations. TXL can be customized to work with different languages, which makes it a very general tool. However, since only the syntax of the transformed languages is defined, transformations generally cannot know about type or binding rules, which makes it difficult to analyze transformations and to guarantee safety properties other than syntactic correctness. In particular, none of the mentioned approaches except MetaML and *CYNTHIA* can guarantee type correctness of the transformed programs. From this point of view, it is interesting to look at Bjørner's approach [2], who defines a simple two-level lambda calculus that offers constructs to generate and to inspect (by pattern matching) lambda calculus terms . 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.

4 Design of an Update Programming Language

From our experience we have found that the following three requirements are crucial for an update (or meta) programming language:

- Context-dependent substitutions.
- Combinations of basic updates.

- Foolproof and convenient handling of bindings.

Context-dependent substitutions can be conveniently described by rewrite rules. However, rules comprise only the basis of an update language. In order to build more complex update programs we need combinators for updates. Examples of indispensable combinators are *alternation* to build alternatives of updates and *recursion* to move updates to arbitrary places in a program. The handling of bindings pertains in particular to high-level programming languages and leads to the concepts of fresh variables and *scope update*, which we discuss in some more detail in the following.

The key idea of our approach 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 (and possibly an update of the symbol itself, which basically means to rename the symbol). We therefore consider the available forms of symbol definitions more closely. In general, a definition has the following form:

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

where v is a symbol (variable) being defined, d is the defining expression for v, 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 just as the *definition* (of v).

Obviously, `let` expression fit this schema of a definition, but also β-redexes have the shape of a definition since a (non-recursive) `let` v`=`d `in` e is just an abbreviation for $(\lambda v.e)\ d$. Moreover, constructor definitions of data type definitions fit this schema with v being the constructor name, d being the type of the constructor, and e being the expression in which the constructor definition is visible. Finally, a pattern/expression pair as found in function definitions or `case` alternatives can also be viewed as a definition. We borrow the term *match rule* from Standard ML [14] for this construct. In a match rule, v is a constructor,[1] d is empty, and e is the RHS. Since d is empty, we essentially represent a match rule by a lambda abstraction $\lambda v.e$. In all cases, possible parameters of v can be represented by corresponding lambda-abstractions in d (or in e in the case of a match rules).

In the next section we will define a general update operation that can be used to update all different kinds of definitions.

Having identified the essential parts of an update calculus is not sufficient to obtain a usable update language. Basically, we need additional syntactic convenience. This is comparable to lambda calculus, which is itself expressive enough, but is hardly ever used as a functional programming language. In contrast, languages like Haskell or ML provide a lot of syntactic sugar to make programming more convenient. A more profound contribution of ML and Haskell are their strong static type systems that help to spot programming errors early. In fact, one of the motivating design criteria for the proposed update language was a type system that can guarantee type correctness (at least to a certain degree) of the generated object programs. For our update language we have identified the following key issues that have considerably influenced the design:

- An economic notation for rules (factored rules)
- Keyword-based syntactic variants of scope updates for specific contexts
- A strong type system

The first two items have been incorporated into the syntax of the update language to be defined in Section 6. The last item is briefly discussed in the next section. A full account is given in a separate paper [8].

[1] In general, a pattern can contain subpatterns, but for simplicity we assume here that this is not the case. In any case, subpatterns can be always eliminated by introducing fresh variables and a corresponding additional `case` expression on the RHS; see, for example, the corresponding transformations described in the Haskell Report [15].

5 The Core Update Calculus

The definition of the update calculus builds on an object language. We use e to range over expressions of the object language and p to range over *patterns*. Patterns comprise meta variables (m) and expressions that do not introduce bindings.

5.1 Update Operations

The update calculus is based on rewrite rules of the form $l \rightsquigarrow r$ where l and r are patterns and where l is linear, that is, l does not contain any variable twice. A rule $l \rightsquigarrow r$ is applied to an expression e by matching l against e, which, if successful, results in a binding σ for the variables in l. The result of the update operation is $\sigma(r)$. To ensure that $\sigma(r)$ is an expression, we require that $MV(r) \subseteq MV(l)$ where MV denotes the set of meta-variables contained in a pattern. If l does not match e, the update described by the rule is not performed, and e remains unchanged.

Complex updates can be built from rules by alternation, recursion, and scope updates.

The *alternation* of two updates u_1 and u_2 is written as $u_1\,;u_2$. The semantics of such an updates is that u_1 is tried to be applied first. Only if u_1 is not applicable, the alternative update u_2 is tried.

Recursion is needed to move updates arbitrarily deep into expressions. We use a recursion operator $\downarrow$ that causes its argument update to be applied (in a top-down manner) to all subexpressions. Different recursion strategies have been discussed in detail elsewhere (for example, [26]). For the purpose of this paper, considering a simple top-down strategy is sufficient, in particular, because recursion is not offered explicitly to the user, but is applied implicitly in scope updates (see below). Simple top-down recursion might not be appropriate always and we might need other recursion operators to be able to express certain updates, however, a detailed discussion of this aspect is beyond the scope of this paper.

The *scope update* facilitates the change of each element of a definition `let` v`=`d `in` e. The update of the symbol v is given by a rule that can rename v; the updates for the definition and the scope are given by arbitrary update expressions. We use the syntax $\{v \rightsquigarrow v' : u_d\}u_e$ for an update that renames v to v', changes v's definition by u_d, and all of its uses by u_e. (We also call $v \rightsquigarrow v'$ the *binding update*, u_d the *definition update*, and u_e the *use update*.) Whereas u_d is only applied to the root of the definition, u_e is always applied recursively. To account for recursive `let` definitions, we apply u_e also recursively to the result obtained by the update u_d. Two special cases of the scope update are obtained if either v' or v are missing: (1) The update $\{\rightsquigarrow v = d\}u$ introduces of a new binding for the variable v and applies u; when $\{\rightsquigarrow v = d\}u$ is applied to an

expression e, it creates the expression `let` v`=`d `in` e' where e' is the result of applying u to e. Note that d can be omitted, in which case the application of $\{\rightsquigarrow v\}u$ to e yields $\lambda v.e'$. (2) The update $\{x\rightsquigarrow e\}u$ applies either to `let` expressions `let` v`=`d `in` e' or lambda abstractions $\lambda v.e'$. It deletes the binding or lambda abstraction for v and applies u to the body e'. The expression e is required in the update and is used to replace all possibly remaining occurrences of v in the result obtained by u.

Surprisingly, we do *not* need an operation to generate fresh variables. Since fresh variables are needed only in two situations, namely, when renaming variables or when creating new definitions, we can integrate the generation of fresh variables into the semantics for these update operations and do not need a separate combinator for it. When an update $\{v\rightsquigarrow w\!:\!u_d\}u_e$ or $\{\rightsquigarrow w\!=\!d\}u$ is applied to an expression e that does not contain the variable w, then w is literally used in the updated expression. Otherwise, w will be renamed (for example, by adding primes or numbers) until an unused variable is found, and the renamed variable is used in u. It has repeatedly been claimed that variable names do not really matter in program transformations and meta-programming [20, 13, 17], but this is not true in our application: when a programmer wants to specify a renaming, she expects the chosen name to appear in the updated program. Should the name already be in use, a related name should be chosen (and possibly a comment should be inserted explaining the use of the different variable) to make program updates transparent.

Finally, we also have a "no update" operation ι that performs no update at all and serves as a unit of the update language. The syntax of updates is shown in Figure 1. In scope updates we use x to range over object variables (v) and meta variables (m).

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
	|	$u\,;u$	Alternative
	|	$\downarrow u$	Recursion

Figure 1. Core update calculus.

Let us consider some examples. The renaming of a function would be expressed by the scope update:

$$\{\text{`}\texttt{f}\rightsquigarrow\text{`}\texttt{g}\!:\!\iota\}\text{`}\texttt{f}\rightsquigarrow\text{`}\texttt{g}$$

Renaming the first parameter of a function `f` can be achieved by a nested scope update, such as:

$$\{\text{`}\texttt{f}\rightsquigarrow\text{`}\texttt{f}\!:\!\{\texttt{x}\rightsquigarrow\text{`}\texttt{y}\!:\!\iota\}\texttt{x}\rightsquigarrow\text{`}\texttt{y}\}\iota$$

The "renaming" `'f`$\rightsquigarrow$`'f` leaves the name of the function `f` unchanged. The update $\{\texttt{x}\rightsquigarrow\text{`}\texttt{y}\!:\!\iota\}\texttt{x}\rightsquigarrow\text{`}\texttt{y}$ will be applied to `f`'s definition and matches if `f`'s definition is given by a `let` expression or a lambda abstraction. In either case the bound variable will be matched by the meta variable `x` and will be renamed to `y`. The use update `x`$\rightsquigarrow$`'y` ensures that all uses of `x` in the body of the `let` expression or lambda abstraction will be renamed to `y`, too. The semantics ensures that `y` will be properly renamed should it conflict with other already bound variables.

We sometimes use the following abbreviations to make core calculus expressions more readable:

- Trivial rules, such as `'f`$\rightsquigarrow$`'f`, can also be written as `'f` whenever it is clear from the context that the single symbol represents a rule.
- We may omit "$:\iota$" from scope updates.

With these two syntactic simplifications, the function renaming, respectively, function parameter renaming, can be written more concisely as:

$$\{\text{`}\texttt{f}\rightsquigarrow\text{`}\texttt{g}\}\text{`}\texttt{f}\rightsquigarrow\text{`}\texttt{g}$$

and

$$\{\text{`}\texttt{f}\!:\!\{\texttt{x}\rightsquigarrow\text{`}\texttt{y}\}\texttt{x}\rightsquigarrow\text{`}\texttt{y}\}\iota$$

5.2 Update Semantics

We define the semantics of updates by judgments of the form $[\![u]\!]_\rho(e)=e'$. The semantics of rules and operations, such as alternative and recursion, is straightforward and has been described extensively in the literature on rewriting. Therefore, we describe here only the semantics of scope updates. In the semantics rules we make use of the following notational conventions.

The set ρ contains variables that are bound by enclosing expressions. For example, suppose we apply an update u recursively to the expression $\lambda v.e$, which means to apply u to e (and possibly to its subexpressions). In that case ρ will be extended by v to ensure that fresh variables (which might be introduced by u) are different from v to prevent them from being illegally bound by enclosing binders like "$\lambda v.$". 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. This is a variable that is neither bound in e nor is contained in ρ. If v has this property, it will be chosen, otherwise an appropriate name will be constructed. $x\succcurlyeq_\sigma v$ means that (the object or meta variable) x matches the (object) variable v under the substitution σ. There are two possible cases: (i) x is an object variable. In this case, x must be equal to v to match v and σ is empty. (ii) x is a meta variable, say m. In this case, x always matches v under the substitution $\sigma=\{m\mapsto v\}$. Finally, we write $u[x:=v]$ for the update u with all occurrences of x replaced by v.

In Figure 2 we show three rules that define the semantics of scope updates: (i) for a scope-changing update applied to a `let` expression (the application to β-redexes and lambda abstractions is similar), (ii) for an insert-scope update yielding a `let` expression (the case when the defining expression is missing is similar and yields a lambda abstraction), and (iii) for a delete-scope update applied to a `let` expression (again the application to a lambda abstraction is similar).

$$\frac{x\succcurlyeq_\sigma v \qquad \sigma(x')\rangle^{e\ d}_\rho w \qquad \rho'=\rho\cup\{w\} \qquad u'_d=(u_d[x:=v])[x':=w] \qquad u'_e=(u_e[x:=v])[x':=w] \qquad [\![\downarrow u'_e]\!]_{\rho'}([\![u'_d]\!]_{\rho'}(d))=d' \qquad [\![\downarrow u'_e]\!]_{\rho'}(e)=e'}{[\![\{x\rightsquigarrow x'\!:u_d\}u_u]\!]_\rho(\texttt{let}\ v\texttt{=}d\ \texttt{in}\ e)=\texttt{let}\ w\texttt{=}d'\ \texttt{in}\ e'}$$

$$\frac{v\rangle^e_\rho w \qquad [\![\downarrow u[v:=w]]\!]_{\rho\cup\{w\}}(e)=e'}{[\![\{\rightsquigarrow v\!=\!d\}u]\!]_\rho(e)=\texttt{let}\ w\texttt{=}d\ \texttt{in}\ e'}$$

$$\frac{x\succ v \qquad [\![\downarrow(u[x:=v]\,;v\rightsquigarrow e_x)]\!]_\rho(e)=e'}{[\![\{x\rightsquigarrow e_x\}u]\!]_\rho(\texttt{let}\ v\texttt{=}d\ \texttt{in}\ e)=e'}$$

Figure 2. Semantics of scope updates.

Note that the freshness precondition in the first rule requires that

w is fresh with respect to *e and d*, that is, w must not be bound in either expression. A complete semantics definition can be found in [8].

5.3 Type Change System

The type system for the update language is designed to find all possible type changes (δ) that an update can cause to an *arbitrary* object program. If these type changes cover each other appropriately, the update can be regarded to be safe with respect to preserving the type correctness of object programs. A simple type change is given by a pair of types and is written as $t \rightsquigarrow t'$. For typing the alternation combinator we also need alternative type changes, which are written as $s \rightsquigarrow s' | t \rightsquigarrow t'$. So in general, a type change can be thought of as a set of simple type changes. To obtain a precise description of type changes that can occur through the use of recursively performed updates, we have to qualify types and type changes by what we call *type contexts*. For brevity we ignore type contexts in the following discussion because the ideas of type-change inference, normal form, and safety of updates can be also illustrated without them. A detailed description of type contexts and their use in the description of type changes can be found in [9].

Type-change judgments are of the form $\Delta \triangleright u :: \delta$ where Δ is a set of *type-change assumptions*, which are basically of the form $x \rightsquigarrow x' :: t \rightsquigarrow t'$ and express the fact that x is renamed to x' and that the type t of x is changed to t'. (We also have assumptions $v :_r t$ for newly introduced variable and $x :_\ell t$ for variables to be deleted). We can define projection onto the left or right part of a type change assumption (written as Δ_ℓ and Δ_r) which both yield sets of ordinary type assumptions (such as $x : t$ and $x' : t'$) that can be used by the type inference of the object language. Then we can define the type change for update rules and scope updates as shown in Figure 3.

$$\text{RUL}\ \frac{\Delta_\ell \vdash e : t \qquad \Delta_r \vdash e' : t'}{\Delta \triangleright e \rightsquigarrow e' :: t \rightsquigarrow t'}$$

$$\text{ALT}\ \frac{\Delta \triangleright u :: \delta \qquad \Delta \triangleright u' :: \delta' \qquad \delta \stackrel{=}{\equiv} \delta'}{\Delta \triangleright u\,;\, u' :: gen(\delta, \delta')}$$

$$\text{CHG}\ \frac{\Delta, x \rightsquigarrow x' :: t \rightsquigarrow t' \triangleright u_d :: t \rightsquigarrow t' \quad \Delta, x \rightsquigarrow x' :: t \rightsquigarrow t' \triangleright u_u :: \delta}{\Delta \triangleright \{x \rightsquigarrow x' : u_d\} u_u :: \delta}$$

$$\text{INS}\ \frac{\Delta, w :_r t \triangleright u :: \delta}{\Delta \triangleright \{\rightsquigarrow w\} u :: t \underset{r}{\rightarrow} \delta}$$

Figure 3. Type change system (excerpt).

The first rule RUL shows that the type-change system is built on the type system of the object language: to determine the type change of an update rule, we have to determine the types of the rule's left-hand side and right-hand side with respect to the corresponding projections of type-change assumptions.

The notation $t \underset{r}{\rightarrow} \delta$ used in the rule INS denotes the extension of a type change's result type by a new argument type t. For a simple type change we have: $t \underset{r}{\rightarrow} s \rightsquigarrow s' = s \rightsquigarrow t \rightarrow s'$; for a type alternative we get: $t \underset{r}{\rightarrow} \delta | \delta' = t \underset{r}{\rightarrow} \delta | t \underset{r}{\rightarrow} \delta'$.

The relationship $\delta \stackrel{=}{\equiv} \delta'$ expresses that one of the type changes is an *applicative instance* of the other, which is true if argument and result type of one type change are applicative instances of argument and result type of the other type change, where a type t is an applicative instance of type u if $u = t$ or $u = s \rightarrow t$ (for some type s). Two updates that have applicative-instance compatible type changes can be considered well typed in an alternative because one update has just a more specific, but compatible, type change than the other. The expression $gen(\delta, \delta')$ then selects the more general of the two type changes as a representative type change for the alternative update.

The Hindley/Milner type system for lambda calculus has the strong property that well-typed programs cannot produce a runtime type error. A corresponding property for the type-change system of the update calculus is that updates for which a type change can be inferred do not produce object programs containing type errors. To ensure this kind of safety, we need an additional structural constraint on updates which we will describe next.

The structural constraint consists of two parts:

(i) 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. This rule prevents ill-typed applications of changed symbols as well as unbound variables.

(ii) 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: $t' \succ t$, that is, t is a type instance of t'. This condition prevents that changed symbols break the well-typing of their contexts.

We say that an update for which a type change can be successfully inferred and that satisfies these two conditions is in *normal form*. We can prove that any scope update in normal form preserves the type correctness of generated/transformed object programs; see [8, 9].

Let us illustrate type-change inference and the normal form by two update examples.

First, we consider an update that generalizes a (value or function) definition by adding a parameter and by replacing an expression in the definition by this parameter. Uses of the function are extended by an application to the abstracted expression.

$$\{\texttt{`f} : \{\rightsquigarrow \texttt{`x}\}\texttt{1} \rightsquigarrow \texttt{`x}\}\texttt{`f} \rightsquigarrow \texttt{`f 1}$$

We use the following two abbreviations in the inference:

$$\Delta_1 = \{\texttt{`f} \rightsquigarrow \texttt{`f} :: \texttt{Int} \rightsquigarrow \texttt{Int->Int}\}$$
$$\Delta_2 = \{\texttt{`f} \rightsquigarrow \texttt{`f} :: \texttt{Int} \rightsquigarrow \texttt{Int->Int}, \texttt{`x} :_r \texttt{Int}\}$$

Now we can conclude according to rule RUL that:

$$\Delta_2 \triangleright \texttt{1} \rightsquigarrow \texttt{`x} :: \texttt{Int} \rightsquigarrow \texttt{Int}$$

Using this fact, we can apply the rule INS to obtain:

$$\Delta_1 \triangleright \{\rightsquigarrow \texttt{`x}\}\texttt{1} \rightsquigarrow \texttt{`x} :: \texttt{Int} \rightsquigarrow \texttt{Int->Int}$$

Again, by applying rule RUL we get:

$$\Delta_1 \triangleright \texttt{`f} \rightsquigarrow \texttt{`f 1} :: \texttt{Int} \rightsquigarrow \texttt{Int}$$

Now we can take the last two facts as premises for the rule CHG, which provides us with the following conclusion.

$$\varnothing \triangleright \{\texttt{`f} : \{\rightsquigarrow \texttt{`x}\}\texttt{1} \rightsquigarrow \texttt{`x}\}\texttt{`f} \rightsquigarrow \texttt{`f 1} :: \texttt{Int} \rightsquigarrow \texttt{Int}$$

We further claim that this update is in normal form, which can be seen as follows. Condition (i) is fulfilled since the type change

for `f` is accompanied by the use update `‘f⇝‘f 1`, whereas condition (ii) is satisfied since the type change for the use update is just `Int⇝Int`.

As a more complex example we consider the introductory example from Section 2 expressed as a core calculus expression u:

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

How this expression can be derived from the HULA update will be demonstrated in Section 6.2. To formally infer the type change for this update, we have to extend the presented type-change system in several ways. First of all, we need a rule to deal with constructor updates, which should be of the following form:

$$\text{CON}\ \frac{\Delta, C_1 \rightsquigarrow C_2 :: t_1 \rightsquigarrow t_2 \triangleright u :: \delta}{\Delta \triangleright \{C_1 \rightsquigarrow C_2 : t_1 \rightsquigarrow t_2\} u :: \delta}$$

Second, the `case` update is translated into the core calculus as a change update for constructors. This works fine with the semantics, but the type-change system needs some modification to deal with this situation.

As we see in the above example, there are two nested updates for constructor `Node`, where the inner resulted from the `case` update. This update should be handled by the type-change system differently than other scope updates, because the type change for the constructor has already been explicitly introduced by the rule CON. It is easy to syntactically identify such updates since the definition update is always ι. Therefore, we can introduce a specialized type-change inference rule for this case:

$$\text{CHG'}\ \frac{\Delta(C) = t \xrightarrow{r} \delta \qquad \Delta \triangleright u :: t \xrightarrow{r} \delta}{\Delta \triangleright \{C : \iota\} u :: \delta}$$

This rule expresses the expectation that the type change inferred for a constructor used in a match rule agrees with the type change explicitly given for the constructor in the type-change environment.

In the following discussion we abbreviate the type-change environment $\{$`Node⇝Node :: a⇝Int->a`$\}$ by Δ. We use u_c to abbreviate the alternative `({Node}(...);{Leaf}Node⇝Node 1)` and u_a for `Node⇝Node 1`.

According to CHG', we can infer the following type changes for the two alternatives:

$\Delta \triangleright$ `{Node}({⇝‘s}Node⇝Node (‘succ ‘s)) :: a⇝a`
$\Delta \triangleright$ `{Leaf}Node⇝Node 1 :: a⇝a`

By applying rule RUL, we also have:

$$\Delta \triangleright \texttt{Node} \rightsquigarrow \texttt{Node 1} :: \texttt{a} \rightsquigarrow \texttt{a}$$

By applying ALT twice, we can infer:

$$\Delta \triangleright u_c\,;u_a :: \texttt{a} \rightsquigarrow \texttt{a}$$

Finally, we can apply rule CON and obtain:

$$\varnothing \triangleright u :: \texttt{a} \rightsquigarrow \texttt{a}$$

This update is also in normal form because (i) the type change for `Node` is accompanied by the rule `Node⇝Node 1` in the use update and (ii) the type change for u is `a⇝a`.

6 The Haskell Update Language

We define the syntax of the update language in Section 6.1. The translation into the core calculus is described in Section 6.2.

6.1 Syntax

The update language builds on the core calculus, in particular, rules and alternation are reused. For syntactics convenience we introduce specialized notations for scope updates and use an economic notation for rules. The general syntactic schema of all scope update constructs is:

$$\textit{cat}\ \textit{bind}\ \texttt{:}\ \textit{def}\ \texttt{in}\ \textit{use}$$

cat marks the syntactic construct to be updated (for example, `con` for a constructor or `fun` for a function definition), *bind* denotes the symbol whose definition and use is being updated and a possible renaming. In general, symbols can introduce bindings for local variables. Therefore, the *bind* part also allows the update of these bindings by renaming symbols or constructors or introducing new or deleting existing symbols. The `:` separates the updates for the bindings from the update of the definition for the updated symbol. Finally, the keyword `in` introduces the update of uses for the updated symbol. This update can be empty. We define that the use part extends as far as possible. The binding update is given by a rule whereas the definition and use part of an update can be given by an arbitrary update expression. In practice, *bind* will be just a name (seldom a renaming) and sometimes followed by an insertion of locally defined variables. Moreover, in most cases, *def* and *use* are given by alternatives of rules or other scope updates. A special syntax is used for `case` updates since these require a list of binding and use updates (without definition updates).

The prevailing part of most updates consists of rules. To make update programs well readable we have therefore thought about what would be the most convenient rule notation. Traditionally, rules are written like $l \rightsquigarrow r$ (see also Section 5). In many cases rules are used to provide context for adding, deleting or replacing a syntactic object, which means that quite frequently parts of l are repeated in r. We have therefore chosen the notation $a\{l/r\}b$ as an abbreviation for $alb \rightsquigarrow arb$. Thus `{x/y}` means replace `x` by `y`. This interpretation was chosen because we obtain as special cases `{x/}` meaning “delete `x` from the context” and `{y}` meaning “insert `x` into the context”, while `{x/y}` still reads like a rule. We say that such a rule is *completely factored* if `x` and `y` do not have a common prefix or suffix.

The syntax of the Haskell update language HULA is defined in Figure 4 and is built on top of the syntax for the manipulated object language Haskell. We require Haskell variables (*hvar*) to be prefixed by a backquote ‘ so that we can use names that begin with a lowercase letter as meta variables (*mvar*), which are variables in HULA. Hence, the syntactic Haskell categories of expressions (*exp*) and types (*type*) are extended to contain quoted variables. We combine the adjusted Haskell expressions and types in the HULA syntactic category *pat* and both kinds of variables as well as Haskell constructors (*hcons*) in the category *sym*.

6.2 Translation into the Update Calculus

The translation of HULA into the update calculus is defined by the function $\mathcal{T}$, which uses four auxiliary functions $\mathcal{T}_{\mathcal{D}}/\mathcal{D}$ and $\mathcal{T}_{\mathcal{U}}/\mathcal{U}$ that deal with the translation of scope updates. The definitions for all these functions are shown in Figure 5.

$$
\begin{array}{lcl}
\mathcal{T}(p\,\{l/r\}\,p') & = & p\,l\,p' \rightsquigarrow p\,r'\,p' \\
\mathcal{T}(\mathit{cat}\ r : u_d\ \texttt{in}\ u_e) & = & \mathcal{T}_{\mathcal{D}}(r, \mathcal{T}(u_d), \mathcal{T}(u_e)) \\
\mathcal{T}(\mathit{cat}\ r : u_d) & = & \mathcal{T}_{\mathcal{D}}(r, \mathcal{T}(u_d), \iota) \\
\mathcal{T}(\texttt{case}\ r_1\ \texttt{->}\ u_1\ \texttt{|}\ \ldots\ \texttt{|}\ r_n\ \texttt{->}\ u_n) & = & \mathcal{T}_{\mathcal{U}}(r_1, \mathcal{T}(u_1))\ ;\ \ldots\ ;\ \mathcal{T}_{\mathcal{U}}(r_n, \mathcal{T}(u_n)) \\
\mathcal{T}(u_1\ ;\ u_2) & = & \mathcal{T}(u_1); \mathcal{T}(u_2) \\
\mathcal{T}(\texttt{(}u\texttt{)}) & = & \mathcal{T}(u) \\
 & & \\
\mathcal{T}_{\mathcal{D}}(\{x/x'\}r^*\{i_1\}r_1^* \ldots r_{n-1}^*\{i_n\}r_n^*, u_d, u_e) & = & \{x \rightsquigarrow x' : \mathcal{D}(r^*, \{\rightsquigarrow i_1\}\mathcal{D}(r_1^*, \ldots, \{\rightsquigarrow i_n\}\mathcal{D}(r_n^*, u_d)))\}u_e \\
\mathcal{T}_{\mathcal{U}}(\{x/x'\}r^*\{i_1\}r_1^* \ldots r_{n-1}^*\{i_n\}r_n^*, u_e) & = & \{x \rightsquigarrow x' : \iota\}\mathcal{U}(r^*, \{\rightsquigarrow i_1\}\mathcal{U}(r_1^*, \ldots, \{\rightsquigarrow i_n\}\mathcal{U}(r_n^*, u_e))) \\
 & & \\
\mathcal{D}(\{x_1/x_1'\} \ldots \{x_n/x_n'\}, u_d) & = & \{x_1 \rightsquigarrow x_1' : \iota\} \ldots \{x_n \rightsquigarrow x_n' : \iota\}u_d \\
\mathcal{U}(\{x_1/x_1'\} \ldots \{x_n/x_n'\}, u_e) & = & \{x_1 \rightsquigarrow x_1' : \ldots \{x_n \rightsquigarrow x_n' : \iota\}u_e \ldots\}\iota
\end{array}
$$

Figure 5. Translation of HULA into the core calculus.

upd	::=	*rule*	factored rule
	|	*cat bind* : *upd* [`in` *upd*]	scope updates
	|	`case` *mrule* { `|` *mrule* }	case update
	|	*upd* `;` *upd*	alternative
	|	`(`*upd*`)`	grouping
mrule	::=	*bind* `->` *upd*	match-rule update
rule	::=	*chng* { *chng* }	update rule
chng	::=	`{`*pat*`/`*pat*`}`	replacement
	|	`{`*pat*[`=` *pat*]`}`	insertion
	|	*pat*	no change
bind	::=	*ren* { *ren* }	binding update
ren	::=	`{`*sym*`/`*sym*`}`	renaming
	|	`{`*sym*`}`	new symbol
	|	*sym*	keep symbol
pat	::=	*exp* | *type*	Haskell objects
sym	::=	*mvar* | *hvar* | *hcons*	variables, constructors
cat	::=	`data` | `con` | `fun`	scope categories

Figure 4. Syntax of the Haskell update language.

The translation of rules and alternatives is rather obvious. The translation of scope updates (and case updates) is complicated by the fact that our rule notation allows the notation of a sequence of nested rules in a linear form (since a *rule* is given by a sequence of *chng*'s). Each such sequence of elementary rules has to be translated into a nested scope update of the core calculus.

We can distinguish two kinds of elementary rules: (i) insertion rules and (ii) replacement and no-change rules, where a no-change rule like `x` is just an abbreviation for the identity replacement rule `{x/x}`. According to this classification we can regard a binding update as a sequence of insertion rules each separated by a (possibly empty) sequence of replacement rules, written as $r^*\{i_1\}r_1^* \ldots r_{n-1}^*\{i_n\}r_n^*$. This view is used in the definition for the functions $\mathcal{T}_{\mathcal{D}}$ and $\mathcal{T}_{\mathcal{U}}$.

The function $\mathcal{T}_{\mathcal{D}}$ is used for translating scope updates, whereas $\mathcal{T}_{\mathcal{U}}$ is used for translating `case` updates. The two functions differ in how they promote definition (u_d) and use updates (u_e) along a sequence of elementary rules: $\mathcal{T}_{\mathcal{D}}$ creates recursively nested definition updates by successively translating elementary rules; $\mathcal{T}_{\mathcal{D}}$ moves u_d downward along the nested rules and leaves u_e on the top level. In contrast, $\mathcal{T}_{\mathcal{U}}$ creates a recursively nested use update; it leaves u_d on the top level and moves u_e downward the nested rules.

The shown translation assumes that rules are completely factored and that the proper distinction between object and meta variables has already been made. We also assume that no-change rules have been expanded into corresponding identity replacements and that keep-symbol rules have been expanded into identity renamings.

To understand how these functions work, it is best to look at some examples. The following update adds a parameter to the function definition for `f` and an argument to all calls to `f`:

```
fun `f {x} : e in `f {3}
```

First, we expand the keep-symbol binding update `` `f `` into the renaming `{`f/`f}` and the no-change rule `e` into the identity replacement `{e/e}`. We also identify `x` as an object variable so that we can use `` `x `` in the following translation. Note that the `` `f `` in the use update is not expanded because it is parsed as a part of the factored rule `` `f {3} `` that will be translated by $\mathcal{T}$.

Now the translation into the core calculus proceeds as follows:

$$
\begin{array}{l}
\mathcal{T}(\texttt{fun `f \{`x\} : e in `f \{3\}}) = \\
\mathcal{T}(\texttt{fun \{`f/`f\} \{`x\} : \{e/e\} in `f \{3\}}) = \\
\mathcal{T}_{\mathcal{D}}(\texttt{\{`f/`f\} \{`x\}}, \mathcal{T}(\texttt{\{e/e\}}), \mathcal{T}(\texttt{`f \{3\}}))
\end{array}
$$

At this point we can apply the first translation rule twice to obtain $\mathcal{T}(\texttt{\{e/e\}}) = \texttt{e} \rightsquigarrow \texttt{e} =: u_d$ and $\mathcal{T}(\texttt{`f \{3\}}) = \texttt{`f} \rightsquigarrow \texttt{`f 3} =: u_e$. We continue by applying the definition for $\mathcal{T}_{\mathcal{D}}$ where x and x' both match `` `f ``, r^* is empty (ε), i_1 matches `{`x}`, and r_n^* is also empty (we have $n = 1$). With $\mathcal{D}(\varepsilon, u) = u$ we obtain:

$$
\begin{array}{l}
\mathcal{T}_{\mathcal{D}}(\texttt{\{`f/`f\} \{`x\}}, u_d, u_e) = \\
\{\texttt{`f} \rightsquigarrow \texttt{`f} : \mathcal{D}(\varepsilon, \{\rightsquigarrow \texttt{`x}\}\mathcal{D}(\varepsilon, u_d))\}u_e = \\
\{\texttt{`f} \rightsquigarrow \texttt{`f} : \{\rightsquigarrow \texttt{`x}\}u_d\}u_e = \\
\{\texttt{`f} \rightsquigarrow \texttt{`f} : \{\rightsquigarrow \texttt{`x}\}\texttt{e} \rightsquigarrow \texttt{e}\}\texttt{`f} \rightsquigarrow \texttt{`f 3} = \\
\{\texttt{`f} : \{\rightsquigarrow \texttt{`x}\}\texttt{e}\}\texttt{`f} \rightsquigarrow \texttt{`f 3}
\end{array}
$$

This example demonstrates how nested binding updates used in scope updates are translated into nested definition updates. The translation is what we expect because the intention was to extend `f`'s definition by a new parameter, and this is exactly what the resulting core-calculus expression achieves.

To understand the need for the functions $\mathcal{T}_{\mathcal{U}}$ and $\mathcal{U}$, consider the translation of the following `case` update.

```
case Node {s} -> Node {1}
```

Again, we first expand the binding update and identify object variables so that we can apply the translation.

$\mathcal{T}$(`case Node {s} -> Node {1}`) =
$\mathcal{T}$(`case {Node/Node} {‘s} -> Node {1}`) =
$\mathcal{T}_{\mathcal{U}}$(`{Node/Node} {‘s}`, $\mathcal{T}$(`Node {1}`))

The first rule for $\mathcal{T}$ gives $\mathcal{T}$(`Node {1}`) = `Node`$\rightsquigarrow$`Node 1`, which we abbreviate by u. Next we apply the definition for $\mathcal{T}_{\mathcal{U}}$. Here x and x' match `Node`, r^* is empty, i_1 matches `{‘s}`, and r_n^* is also empty (again, $n = 1$). With $\mathcal{U}(\varepsilon, u) = u$ we can continue:

$\mathcal{T}_{\mathcal{U}}$(`{Node/Node} {‘s}`, u) =
`{Node`$\rightsquigarrow$`Node:ι}` $\mathcal{U}(\varepsilon$, `{`$\rightsquigarrow$`‘s}` $\mathcal{U}(\varepsilon, u))$ =
`{Node`$\rightsquigarrow$`Node:ι}{`$\rightsquigarrow$`‘s}`u =
`{Node`$\rightsquigarrow$`Node:ι}({`$\rightsquigarrow$`‘s}Node`$\rightsquigarrow$`Node 1)` =
`{Node}({`$\rightsquigarrow$`‘s}Node`$\rightsquigarrow$`Node 1)`

This example demonstrates that nested binding updates used in `case` updates are translated into nested use updates, which makes sense since the symbols introduced in match rules have no definition. Instead the use of the introduced symbols in the right-hand side of the match has to be updated. This is accomplished by the resulting core-calculus expression.

As a slightly larger example we consider how the update from Section 2 is translated by $\mathcal{T}$ into a core calculus expression. As we did in the previous examples, we first expand the binding update and identify object variables:

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

We abbreviate the `case` update by u_c and the alternative update for `Node` extensions by u_a, that is, we consider the translation of the update `con {Node/Node} : {Int} t in` u_c`;` u_a. We obtain:

$\mathcal{T}$(`con {Node/Node} : {Int} t in` u_c`;` u_a) =
$\mathcal{T}_{\mathcal{D}}$(`{Node/Node}`, $\mathcal{T}$(`{Int} t`), $\mathcal{T}$(u_c`;` u_a)) =
$\mathcal{T}_{\mathcal{D}}$(`{Node/Node}`, `t`$\rightsquigarrow$`Int t`, $\mathcal{T}$(u_c`;` u_a))

The rule for `case` yields for $\mathcal{T}(u_c)$:

$\mathcal{T}_{\mathcal{U}}$(`{Node/Node} {‘s}`, $\mathcal{T}$(`Node {‘succ ‘s}`));
$\mathcal{T}_{\mathcal{U}}$(`{Leaf/Leaf}`, $\mathcal{T}$(`Node {1}`))

Similar to the previous example, this expression can be further translated to:

```
{Node}({⇝‘s}Node⇝Node 1);
{Leaf}Node⇝Node 1
```

Therefore, we obtain for $\mathcal{T}(u_c\,;\ u_a)$ the following core calculus expression.

```
({Node}({⇝‘s}Node⇝Node 1);
 {Leaf}Node⇝Node 1);
Node⇝Node 1
```

We can now complete the translation of the complete `con` update as follows.

$\mathcal{T}$(`con {Node/Node} : {Int} t in` u_c`;` u_a) =
$\mathcal{T}_{\mathcal{D}}$(`{Node/Node}`, `t`$\rightsquigarrow$`Int t`, $\mathcal{T}$(u_c`;` u_a)) =

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

By abbreviating the trivial rule `Node`$\rightsquigarrow$`Node`, we finally obtain:

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

7 Conclusions and Future Work

In this paper we have introduced an update language for Haskell called HULA. HULA is based on combinators to build complex updates from basic rewrite rules. An important safety property of HULA is that when a type-safe update in normal form is applied to a type-correct program, it produces another type-correct program. In terms of Milner's slogan, type-safe updates will not compile wrong.

HULA is still limited in its current form. Some of the extensions we plan to work on in future are:

Supporting full Haskell. Currently, we deal only with a subset of Haskell. For example, the treatment of type classes requires an extension of the notion of type change. Then type safety requires also kind safety of updates. Kind changes can be formalized similarly to type changes. There are other issues, such as the treatment of modules, that should be supported, too. Extensions of HULA have to be supported by the core calculus, which might also require extensions in some cases. For example, we have seen that although HULA updates for constructors and `case` expression can be translated into core calculus expressions that are handled well by the semantics, the type-change inference requires new rules.

Generic updates and update libraries. Looking at the introductory example, we can observe a certain general pattern: 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. Such a generic update could be written once and stored in an update library, so that constructor extensions as the one for `Node` can be expressed as applications of such a general operation. Assuming the generic operation is called `extCon`, the `Node` update can then be expressed by:

```
extCon Node Int succ 1
```

which would have exactly the same effect as the update shown in Section 2. In order to facilitate such update functions, HULA has to be extended by function definitions and applications. The update function `extCon` could then be implemented, for example, as follows.

```
extCon c t f e = con c : {t} u in
                     (case c {x} -> c {f x}
                         | _     -> c {e});
                     c {e}
```

Even such an innocent-looking generalization might raise some dif-

ficult issues for the type system. For example, instead of the constructor `Leaf` that was used in the original update program in the second match-rule update, we have to use a wildcard symbol (`_`) in the generic update. Whereas `Leaf` has one particular type, the wildcard has to range over many different constructors of different types.

Conditional type safety. The normal form to guarantee type safety is rather strict so that many useful program updates would not be classified as type safe. It seems that this problem occurs, in particular, for some generic updates. However, in many situations, "complete" type safety is not mandatory. Instead, a form of conditional type safety is sufficient. The second example from Section 2 is not type safe because the update does not update any `Node` outside the `insert` function. But, it is safe as long as the object program contains only the `insert` function. The notion of *conditional safety* in the sense that type safety is preserved only for object programs that satisfy some constraints (such as containing certain functions) is not as strong as unconditional safety, but it is more widely applicable and is still much better than having no information at all.

Improved implementation. We currently have a stable implementation of the core calculus in Haskell. This prototype is to be extended to full HULA and full Haskell. For improved efficiency, we also want to consider translating HULA into a rewriting system, such as Stratego.

Acknowledgments

The authors would like to thank the reviewers for their insightful comments and suggestions that helped to improve the paper.

8 References

[1] ACM. *Communications of the ACM*, volume 44(10), October 2001.

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

[3] 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.

[4] B. Borovanský, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN: A Logical Framework Based on Computational Systems. In *Workshop on Rewriting Logic and Applications*, 1996.

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

[6] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow. TXL: A Rapid Prototyping System for Programming Language Dialects. *Computer Languages*, 16(1):97–107, 1991.

[7] M. Erwig. Programs are Abstract Data Types. In *16th IEEE Int. Conf. on Automated Software Engineering*, pages 400–403, 2001.

[8] M. Erwig. Update your Program? Program Your Update! Technical Report TR02-60-01, Department of Computer Science, Oregon State University, 2002. `http://www.cs.orst.edu/~erwig/papers/up.pdf`.

[9] M. Erwig and D. Ren. An Update Calculus for Type-Safe Program Changes. 2002. submitted for publication. `http://www.cs.orst.edu/~erwig/papers/uc.pdf`.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 1999.

[11] 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.

[12] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *4th Symp. on Practical Aspects of Declarative Languages*, LNCS 2257, pages 137–154, 2002.

[13] D. Miller. Abstract Syntax for Variable Binders: An Overview. In *Computational Logic*, LNAI 1861, pages 239–253, 2000.

[14] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[15] S. L. Peyton Jones, J. Hughes, et al. Report on the Programming Language Haskell 98, 1999. `http://haskell.org/onlinereport`.

[16] F. Pfenning. Logical Frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 21. Elsevier Science Publishers, Amsterdam, NL, 2001.

[17] F. Pfenning and C. Elliott. Higher-Order Abstract Syntax. In *ACM Conf. on Programming Languages Design and Implementation*, pages 199–208, 1988.

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

[19] D. Roberts and J. Brant. Refactoring Tools. In M. Fowler, editor, *Refactoring: Improving the Design of Existing Code*, chapter 14, pages 309–352. Addison-Wesley, Reading, MA, 1999.

[20] T. Sheard. Accomplishments and Research Challenges in Meta-Programming. In *2nd Int. Workshop on Semantics, Applications, and Implementation of Program Generation*, LNCS 2196, pages 2–44, 2001.

[21] W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[22] C. Verhoef. Towards Automated Modification of Legacy Assets. *Annals of Software Engineering*, 9:315–336, 2000.

[23] E. Visser. Strategic Pattern Matching. In *10th Int. Conf. on Rewriting Techniques and Applications*, LNCS 1631, pages 30–44, 1999.

[24] E. Visser. Language Independent Traversals for Program Transformation. In *Workshop on Generic Programming*, 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

[25] E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *12th Int. Conf. on Rewriting Techniques and Applications*, LNCS 2051, 2001.

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

[27] E. Visser, et al. The Online Survey of Program Transformation. `http://www.program-transformation.org/survey.html`.

[28] J. Whittle, A. Bundy, R. Boulton, and H. Lowe. An ML Editor Based on Proof-as-Programs. In *9th Int. Symp. on Programming Language Implementation and Logic Programming*, LNCS 1292, pages 389–405, 1997.