Science of Computer Programming 112 (2015) 145–169

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Type-changing rewriting and semantics-preserving transformation

Sean Leather [a,*], Johan Jeuring [a,b], Andres Löh [c], Bram Schuur [a]

[a] Utrecht University, The Netherlands
[b] Open University, The Netherlands
[c] Well-Typed LLP

ARTICLE INFO

Article history:
Received 26 June 2014
Received in revised form 1 July 2015
Accepted 24 July 2015
Available online 13 August 2015

Keywords:
Automatic program transformation
Type-changing rewriting
Semantics-preserving program transformation
Type-and-transform systems

ABSTRACT

We have identified a class of whole-program transformations that are regular in structure and require changing the types of terms throughout a program while simultaneously preserving the initial semantics after transformation. This class of transformations cannot be safely performed with typical term rewriting techniques, which do not allow for changing the types of terms.

In this paper, we present a formalization of type-and-transform systems, an automated approach to the whole-program transformation of terms of one type to terms of a different, isomorphic type using type-changing rewrite rules. A type-and-transform system defines typing and semantics relations between all corresponding source and target subprograms such that a complete transformation guarantees that the whole programs have equivalent types and semantics. We describe the type-and-transform system for the lambda calculus with let-polymorphism and general recursion, including several examples from the literature and properties of the system.

1. Introduction

Program improvement sometimes involves large, homogeneous changes that are not intended to modify program functionality (other than, perhaps, performance). For example, a programmer might rename variables, reorganize code, or update code to use a new library API. Of course, these changes can still introduce unwanted errors into a program. Consequently, programmers often use tools to help automate common patterns of change such as refactoring [8]. Compilers or interpreters may also be employed for large changes such as optimization without necessitating programmer intervention. In functional programming, term rewriting [1] can be used to safely change programs with simple rewrite rules.

Many approaches to automated semantics-preserving program improvement only allow type-preserving updates to code. This is only natural: in a statically typed programming language, type safety is a prerequisite for a working program. Replacing one term with another of a different type challenges the effort of guaranteeing the preservation of semantics between the terms. Some type-changing rewrites may be straightforward: adding a parameter to a function, for example. Other changes are not obvious: changing one string type to a different string type, in which the APIs of the two types are not equivalent. A completely transformed program should work as before, i.e. the strings are still strings. However,

* Corresponding author.
E-mail addresses: s.p.leather@uu.nl (S. Leather), j.t.jeuring@uu.nl (J. Jeuring), andres@well-typed.com (A. Löh), bramschuur@gmail.com (B. Schuur).

http://dx.doi.org/10.1016/j.scico.2015.07.009
0167-6423/

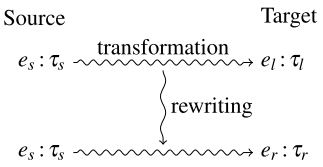

Fig. 1. Diagram of the relationship between transformation and rewriting.

the evaluation may now be more efficient. Or, for example, the program now supports Unicode characters whereas before the encoding was ASCII. Our focus is the class of transformations between isomorphic types with possibly different APIs.

In this paper, we discuss a foundation for certain automated semantics-preserving and type-changing program transformations. We use purely functional programming languages with strong, static type systems. Such languages allow us to utilize the type system for safety as well as driving change throughout the program. By disallowing or isolating side effects, such languages also simplify the proof of semantics preservation. Our object language is the lambda calculus with let-polymorphism and general recursion.

A *type-and-transform system* defines, for a given language, how to relate two programs such that all "unresolved" term and type changes are identified and can (eventually) be resolved resulting in the programs being semantically equivalent. A type-and-transform system specifies the structure of a *transformation*[1] that relates one typed program (the source) to another (the target). A target is actually the possibly modified source. A type-and-transform system also specifies how a program can be modified with a *typed rewrite rule*, an extension of the usual term rewrite rule that can, under certain conditions, impose a change of type between its left-hand side (lhs) and right-hand side (rhs) patterns.

A transformation reflects the structure of the source term, preserving both the syntactic relation of corresponding subterms in the source and target and the typing relation of those subterms. A transformation also records all rewritings to the target by an associated set of typed rewrite rules. A *complete transformation*[2] is a transformation with the same source and target types and equivalent semantics, even though the programs may differ syntactically.

Fig. 1 provides a visualization of the connections between transformation and rewriting. The diagram is split vertically to position the parts relevant to the source program on the left and the target program on the right. A program such as $e_s : \tau_s$ represents the term e_s – in this case, the source term – with its type τ_s. Transformations are horizontal, indicating the relation between source and target, and use an $\rightsquigarrow$ arrow. Applying a typed rewrite rule is a vertical step from one transformation to another with an $\leadsto$ arrow.[3] With typed rewriting, the target term and type can change; however, the transformations "before" and "after" rewriting must each preserve a relation between its respective target and the same source. It is in this sense that typed rewriting relates two transformations rather than two terms, as is typical for term rewriting. In future sections, we will revisit the diagrammatic technique of Fig. 1 to help elucidate the relationships between the components of transformation and rewriting.

The associated set of typed rewrite rules describes all the allowed term and type changes for a transformation. We use two metavariables, $\mathcal{A}$ and $\mathcal{R}$, to indicate the abstraction and representation types, respectively, which are the only types that can be changed. The basic conversion between these types is given by the functions $rep : \mathcal{A} \to \mathcal{R}$ and $abs : \mathcal{R} \to \mathcal{A}$.[4]

In this paper, we focus on types $\mathcal{A}$ and $\mathcal{R}$ that are isomorphic. That is, both of the following equivalences hold:

$$rep \circ abs \equiv id_{\mathcal{R} \to \mathcal{R}} \tag{REP-ABS}$$

$$abs \circ rep \equiv id_{\mathcal{A} \to \mathcal{A}} \tag{ABS-REP}$$

This simplifies the proof of semantics, but it also means many type pairs are not supported.

As an aside, we believe that the isomorphism requirement can be weakened to a retract – that is, only (ABS-REP) would be necessary. A retract would allow transformations between, for example, the types $\mathcal{A} = String$ and $\mathcal{R} = String \to String$, which do not have an isomorphism (see Section 1.1 for why). One of the authors has already shown the retract requirement to some extent. In a master's thesis, Schuur [29] demonstrated a type-and-transform system for the simply typed lambda calculus using a logical relation as proof technique. There is a precedence [27] for using logical relations for more interesting languages such as ours, which has general recursion and polymorphism. We will explore this in future work, but we feel that this paper stands well on its own as an introduction to and foundation for type-and-transform systems.

[1] With apologies for the abuse of terminology, we have borrowed the terms "transformation" and "rewrite," among others, and given them specific meanings that differ from those in other contexts.

[2] This is not related to "completeness" but rather to a subset of transformations obeying certain properties described in Section 6.

[3] The intuition behind the arrows is that, where rewriting is a change or a "bump in the road" ($\leadsto$), a transformation may include a sequence of rewrites or multiple bumps ($\rightsquigarrow$).

[4] We adopted the use of $\mathcal{A}$/*abs* and $\mathcal{R}$/*rep* from Hughes [17].