Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 322 (2016) 19–34

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Incremental Rebinding with Name Polymorphism [1]

Davide Ancona[a,2] Paola Giannini[b,3] Elena Zucca[a,4]

[a] *DIBRIS, Università di Genova, Italy*

[b] *CS Institute, DISIT, Università del Piemonte Orientale, Alessandria, Italy*

Abstract

We propose an extension with *name variables* of a calculus for incremental rebinding of code introduced in previous work. Names, which can be either constants or variables, are used as interface of fragments of code with free variables. Open code can be dynamically rebound by applying a *rebinding*, which is an association from names to terms. Rebinding is *incremental*, since rebindings can contain free variables as well, and can be manipulated by operators such as overriding and renaming. By using name variables, it is possible to write terms which are parametric in their nominal interface and/or in the way it is adapted, greatly enhancing expressivity. The type system is correspondingly extended by *constrained name-polymorphic types*, where simple inequality constraints prevent conflicts among parametric name interfaces.

Keywords: open code, incremental rebinding, name polymorphism, metaprogramming

1 Introduction

Our previous work [1,2] smoothly integrates static binding of the simply-typed lambda-calculus with a mechanism for dynamic and incremental rebinding of code. Fragments of open code to be dynamically *rebound* are values. Rebinding is done on a *nominal* basis, that is, free variables in open code are associated with *names* which do not obey α-equivalence. Moreover, rebinding is *incremental*, since rebindings, which are associations between names and terms, can in turn contain free variables to be rebound. Rebindings are first class values, and can be manipulated by operators such as overriding and renaming.

In this paper, we propose an extension of this previous work which supports, besides name constants, *name variables*, making it possible to write terms which are parametric in their nominal interface and/or the way it is adapted. For instance,

[1] This work has been partially funded by "Progetto MIUR PRIN CINA Prot. 2010LHT4KM".

[2] Email: davide.ancona@unige.it

[3] Email: giannini@di.unipmn.it

[4] Email: elena.zucca@unige.it

http://dx.doi.org/10.1016/j.entcs.2016.03.003

it is possible to write a term which corresponds to the selection of an arbitrary component of a module. We summarize here below the language features.

- *Unbound terms*, of shape $\langle x_1 \mapsto X_1, \ldots, x_m \mapsto X_m \mid t \rangle$ are values representing "open code". That is, t may contain free occurrences of variables $x_1, \ldots, x_m$ to be dynamically bound through the global nominal interface $X_1, \ldots, X_m$. To be used, open code should be combined with a *rebinding* $X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$.
- Rebinding application is *incremental*, that is, an unbound term can be partially rebound, and a rebinding can be open in turn. For instance, the term $\langle x{\mapsto}X, y{\mapsto}Y \mid x{+}y \rangle$ can be combined with the rebinding $\langle y{\mapsto}Y \mid X{\mapsto}y, Z{\mapsto}y \rangle$, getting $\langle y \mapsto Y, y' \mapsto Y \mid y'{+}y \rangle$. This makes possible code specialization, similarly to what partial application achieves for positional binding.
- Rebindings are first-class values as well, and can be manipulated by operators such as overriding and renaming.
- A *name* X can be either a *name constant* N or a name variable α, and *name abstraction* $\Lambda\alpha.t$ and *name application* $t\ X$ can be used analogously to lambda-abstraction and application to define and instantiate name-parametric terms.

The type system in [2], supporting both *open* (*non-exact*) and *closed* (*exact*) types for rebindings, is correspondingly extended to handle name variables. Notably, types are extended with *constrained name-polymorphic types* of shape $\forall\alpha{:}c.\,T$, where c is a set of *inequality constraints* $X{\neq}Y$ among names. Such constraints are necessary to guarantee that for each possible instantiation of α we get well-formed terms and types. For instance, the term $\Lambda\alpha{:}\alpha \neq N.\langle \mid N{:}\texttt{int} \mapsto 0, \alpha{:}\texttt{int} \mapsto 1 \rangle$ is a rebinding parametric in the name of one of its two components, which, however, must be different from the constant name N of the other component.

In the rest of this paper, we first provide the formal definition of an untyped version of the calculus (Section 2), followed by some examples showing its expressive power (Section 3). We then define a typed version of the calculus (Section 4), for which we state a soundness result. We show typing examples in Section 5, and finally in the Conclusion we discuss related and future work.

2 Untyped calculus

The syntax and reduction rules of the untyped calculus are given in Figure 1, where we leave unspecified constructs of primitive types such as integers, which we will use in the examples. We assume infinite sets of *variables* x, *name constants* N and *name variables* α. We use X, Y to range over *names* which are either name constants or name variables.

We use various kinds of sequences which represent finite maps: *unbinding maps* u from variables to names, *rebinding maps* r from names to terms, *renamings* σ from names to names, and *substitutions* s from variables to terms. We assume that order and repetitions are immaterial in such sequences. Moreover, in a term t which is well-formed, written $\vdash t$, they actually represent maps, e.g., in $X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$, if $X_i = X_j$ then $t_i = t_j$. Hence, we can use the following notations: *dom* and *rng* for the domain and range, respectively, $u_1 \circ u_2$ for map composition, assuming $rng(u_2) \subseteq dom(u_1)$, (u_1, u_2) for the union of two maps with disjoint