SIMPLE — Typed — Static

Grigore Roşu and Traian Florin Şerbănuţă (grosu,tserban2@illinois.edu)
University of Illinois at Urbana-Champaign

Abstract

This is the 2 definition of the static semantics of the typed SIMPLE language, or scalar-mode, a type system for the typed SIMPLE language. The type system is a simple type system, with subtyping, that ensures that the dynamic semantics will not get stuck on ill-typed terms of the language PL and A0 definitions. We have only focus on the new and interesting problems raised by the introduction of subtyping, and we do not discuss the standard type system for the simply typed lambda calculus.

When designing a type system for a language, we want to what principles, we have to decide upon the intended typing policy, and what properties we want to prove about the type system. In our case, we want to prove that well-typed terms in the scalar-mode program which don't have a stuck form (of should no other, function) do not get stuck (progress). We should no allow functions where type expects them to return a value (say an int) to use a plain "return;" statement, which returns no value. We want to prove that if a term has type T, then it will evaluate to a value of a subtype of type T (preservation). We want to prove that, for each term and type subtype to be in preservation or possibly, that it, is accept as more programs that do not get stuck. We want to prove that if a term has type T, then it will evaluate to a value of a subtype of type T (preservation). On the other hand, you want your type system to have a reasonable performance when implemented. We will keep both the dynamic semantics and the type system simple, and we will not discuss any implementation issues. Nevertheless, we will publish alternative rules so as not expected to be feasible in general. A simple algorithm when designing typing policies is to ensure that the dynamic semantics will not get stuck on well-typed terms, and to make the type system as simple as possible.

Below we give the 2 type system of SIMPLE formally, we discuss, informally, the intended typing policy.

- Function pointers should contain only (typed) 1 function. Instead the (typed)SIMPLE semantics will get stuck on any program that tries to call a function pointer.
- Each primitive value has its own type, which can be: int, bool, or string. There is also a type void for uninitialized values, for example for the result of a function invocation in return or value that only be used for its side effects, like a function call.
- The syntax of untyped SIMPLE is extended to allow type declarations for all the variables, including array variables. This is done as in C/Java/C#: for example, `int a;` or `int arr[10];` or `int*(int,int) f;` or `int(int,int) g(int,int) h;` The syntax of untyped SIMPLE is also extended to allow type declarations for functions, for example `int f(int,int);` (a 2-dimensional array) are various separated arguments, although they have the array of array semantics.
- Functions are also typed in a Pascal style. However, cases in SIMPLE where functions in the passed to and returned from other functions are allowed. We will not allow nested functions, but we will allow pointers to functions of arbitrary types, but will separate the argument types with commas. For example, a function returning an array of bool's dimension 10 and takes as argument an array of 10 integer-argument function returning an integer is declared using a variant of the form:

```
bool[] f(int,int*(int,int)*(int,int)*int,int,int,
```