Abstract interface behavior of an object-oriented language with futures and promises

3. September 2007

Erika Ábrahám[1], and Immo Grabe[2], and Andreas Grüner[2], and Martin Steffen[3]

[1] Albert-Ludwigs-University Freiburg, Germany
[2] Christian-Albrechts-University Kiel, Germany
[3] University of Oslo, Norway

1 Motivation

How to marry concurrency and object-orientation has been a long-standing issue; see e.g., [2] for an early discussion of different design choices. Recently, the thread-based model of concurrency, prominently represented by languages like *Java* and *C#* has been criticized, especially in the context of *component-based* software development. As the word indicates, components are (software) artifacts intended for composition, i.e., *open* systems, interacting with a surrounding environment. To compare different concurrency models on a solid mathematical basis, a semantical description of the interface behavior is needed, and this is what we do in this work. We present an *open semantics* for a core of the *Creol* language [4,7], an object-oriented, concurrent language, featuring in particular asynchronous method calls and (since recently [5]) *future*-based concurrency.

Futures and promises A *future,* very generally, represents a result yet to be computed. It acts as a proxy for, or reference to, the delayed result from a given sequential piece of code (e.g., a method or a function body in an object-oriented, resp. a functional setting). As the client of the delayed result can proceed its own execution until it actually *needs* the result, futures provide a natural, lightweight, and (in a functional setting) transparent mechanism to introduce parallelism into a language. Since its introduction in *Multilisp* [6][3], futures have been used in various languages, including Alice ML, E, the ASP-calculus, Creol, and others more. A *promise* is a generalization insofar that the reference to the result on the one hand, and the code responsible to calculate the result on the other, are not created at the same time; instead, a promise can be created independently and only later, after possibly passing it around, the promise is bound to the code (one also says, the promise is *fulfilled*).

Interface behavior An *open* program interacts with its environment via message exchange. The interface behaviour of such an open program C can be characterized by the set of all those message sequences (= traces) t, for which there *exists* an environment E such that C and E can exchange the messages recorded in t. Thus the interface behaviour abstracts away of any concrete environment. However, it only considers such environments, that are compliant to the language restrictions (syntax, type system, etc.).

Consequently, interactions do not consist of arbitrary message sequences $C \stackrel{t}{\Longrightarrow}$; instead we consider the behavior $C \parallel E \stackrel{t}{\underset{\bar{t}}{\Longrightarrow}} \acute{C} \parallel \acute{E}$ where E is an *arbitrary* but *realizable* environment and $\bar{t}$ complementary to t.

To account for the existentially abstracted environment ("there exists an E s.t. ..."), the open semantics is given in an *assumption-commitment* way:

$$\Delta \vdash C : \Theta \stackrel{t}{\Longrightarrow} \acute{\Delta} \vdash \acute{C} : \acute{\Theta}$$

where Δ contains (as an abstract version of E) the *assumptions* about the environment, and dually Θ the *commitments* of the component. Abstracting away also from C gives a language characterization by the set of all possible traces between any component and any environment.

Such a behavioral interface description is relevant and useful for the following reasons. 1) The set of possible traces is more restricted than the one obtained when ignoring the environments. I.e., when *reasoning* about the trace-based behavior of a component, e.g., in compositional verification, with more precise characterization one can carry out stronger arguments. 2) When using the trace description for black-box testing, one can describe test cases in terms of the interface traces and then synthesize appropriate test drivers from it. Clearly, it makes no sense to specify impossible interface behavior, as in this case one cannot generate a corresponding tester. 3) A representation-independent behavior of open programs paves the way for a compositional semantics and allows furthermore optimization of components: only if two components show the same external behavior, one can replace one for the other without changing the interaction with any environment. 4) The formulation gives *insight* into the semantical nature of the language, here, the observable consequences of futures and promises. This helps to compare alternatives, for instance the Creol concurrency model with Java-like threading.

2 Results

We formalize the abstract interface behavior for concurrent object-oriented class-based languages with futures and promises. The long version of the submission includes the following results:

Concurrent object calculus with futures and promises We formalize a class-based concurrent language featuring futures and promises, capturing the core aspects of the *Creol*-language. The formalization is given as a typed, imperative object calculus in the style of [1] resp. one of its concurrent extensions. We present the semantics in a way that facilitates comparison with *Java*'s multi-threading concurrency model, i.e., the operational semantics is formulated so that the multi-threaded concurrency as (for instance) in *Java* and the one based on futures here are represented similarly.

Linear type system for promises Featuring promises, the calculus extends the semantic basis of *Creol* as given for example in [5] (only futures). Promises can refer to a computation with code bound to it later. It is important, that the binding is done at most

once. To guarantee such a *write-once* policy when passing around promises, we refine the type system introducing two type constructors

$$[T]^{+-} \quad \text{and} \quad [T]^{+}$$

representing a reference to a promise that can still be written (and read, and with result type T), resp. a reference with read-only permission. The write-permission constitutes a resource which is consumed when the promise is fulfilled. The resource-aware type system is therefore formulated in a *linear* manner wrt. the write permissions and resembles in intention the one in [8] for a functional calculus with references. Our work is more general, in that it tackles the problem in an object-oriented setting (which, however, conceptually does not pose much complications). It is in addition more general in that we do not give a type system for a closed system, but for an open component. Also this aspect of openness is not dealt with in [5]. Additionally, the type system presented here is simpler as the one in [8], as it avoids the representation of the promise-concept by so-called *handled futures.*

Soundness of the abstractions We show soundness of the abstractions, which includes

- *subject reduction,* i.e., preservation of well-typedness under reduction. Subject reduction is not just proven for a closed system (as usual), but for an open program interacting with its environment. Subject reduction implies
- *absence of run-time errors* such as "message-not-understood", again also for open systems.
- A proof that the characterization of the interface behavior is *sound,* i.e., all interaction behavior which is possible by an actual, concrete environment is included in the abstract interface behavior description.
- for promises: absence of *write-errors,* i.e. the attempt to fulfill a promise twice.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.
3. H. Baker and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12:55–59, 1977.
4. The Creol language. `http:www.ifi.uio.no/~creol`, 2007.
5. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. 18th European Symposium on Programming (ESOP'07)*, 2007. To appear in Springer's LNCS series.
6. R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.
7. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.
8. J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda-calculus with futures. *Theoretical Computer Science*, 2006. Preprint submitted to TCS.