DART

IST-2001-33477

Dynamic Assembly, Reconfiguration and Type-checking

Deliverable no: D1.1

Reports on various subtheories for dynamic assembly

Report Version: first

Report Preparation Date: 2003-01-07

Classification: PP

Deliverable no: D1.1 Due Date: month 12 Delivery Date: month 12

Project Start Date: 01.01.2002 Project Duration: 36 months

Project Coordinator: Univ. di Genova

Project Partners: Heriot-Watt Univ., Imperial College, Univ. di Torino

Project funded by the European Community under the 'Information Society Technologies' Programme (1998–2002)

1 Summary

This deliverable presents the work done in WP1 in the first year of the project. It consists of [ALZ02a, ALZ02b], [DLE02], [MF02], [AFMZ02], [FMP02], [CMSar][1], [BBV02a][2], [BBV02b].

These papers fall within two main strands: the first is oriented towards WP4, the second towards WP5. This introduction summarizes the overall progress; the Appendix explains the relevance of each paper and gives a summary of its contribution.

In the first strand, we have taken Java (and partly C#) as a paradigmatic example of environment supporting notions relevant for dynamic software assembly, such as separate compilation, dynamic loading, binary compatibility. We have analyzed in depth the notion of separate compilation: [ALZ02a] develops a formal framework for the compilation mechanism as it is in Java, [ALZ02b] defines a type system which formalizes true separate compilation for Java in the sense of [Car97]. We have also analyzed dynamic linking: [DLE02] develops a non-deterministic model, which includes the behaviour of Java and C#.

In the second strand, we have focused on two important language features for WP1: staging and mixin modules. Staging supports a fine-grained programming of the temporal aspects through run-time code generation and customization. Mixin modules promote software reuse and incremental programming by combining parametrization, cross-module recursion and overriding. In [MF02, AFMZ02] we have studied how these features interact with computational effects, by considering monadic metalanguages MMML and CMS_{do}, whose operational semantics is given in terms of local simplification (independent from computational effects) and computation (that may cause computational effects).

We have also investigated the usefulness of these features for WAN programming, by considering two extensions of Klaim, an experimental Kernel Language for Agents Interaction and Mobility inspired by the Linda coordination model. In MetaKlaim [FMP02] staging provides an ideal tool for describing customization and combination of software components. In MoMi [BBV02a, BBV02b] mixins provide a natural approach to mobile object-oriented code. These two extensions are of interest also for WP5, since the input/read primitives exploit (pattern matching and) dynamic typing to select tuples from a tuple space.

[1]This work was started before DART, but the final version was written within the project.

[2]This work was started before DART, but the final version was written within the project.

2 Appendix

2.1 Separate compilation and Java [ALZ02a, ALZ02b]

One of the aims in WP1 is to provide formal foundations for the process of separate compilation of fragments. Though Java is considered a paradigmatic example of language supporting separate compilation, the Java compilation process does not really correspond to the notion of "true" separate compilation, which has been nicely clarified and formalized in a simple case in the seminal paper [Car97]. Indeed, the compilation of a fragment f which depends on $f_1, \ldots, f_n$ cannot be performed in absence of their code (either source or binary); moreover, if (some of) $f_1, \ldots, f_n$ are only available in source form, then Java compilers enforce their typechecking and generation of the corresponding code, too. No formal description is provided for this process and the rules used by standard compilers are quite intricate.

In [ALZ02a], we model the overall Java compilation process via the formal notion of *compilation schema*, which defines a *source type judgment* and a *binary type judgment* modeling typechecking, in a given type environment, of source code and binary code, respectively, a *dependency function* modeling the fact that typechecking of a fragment may enforce typechecking of other fragments, and a *type extraction function*, modeling the fact that type information for a fragment cannot be provided separately from code. We consider three different compilation schemata for Java: a *minimal* schema which corresponds to true separate typechecking, that used by standard Java compilers and finally a *type safe* compilation schema.

In [ALZ02b], we define a type system modeling true separate compilation for a small but significant Java subset, in the sense that a single class declaration can be intra-checked (following the Cardelli's terminology) and compiled providing a minimal set of type requirements on missing classes. These requirements are specified by a local type environment associated with each single class. We also provide formal rules for static inter-checking and relate our approach with compilation of closed programs, by proving that we get the same results.

2.2 Flexible models for dynamic linking [DLE02]

Another important operation of manipulation of fragments is dynamic linking. Dynamic linking supports flexible code deployment and update: instead of fully linking code before execution, further code is linked on the fly, as needed. One question connected to dynamic linking is the choice of components to be linked, especially if there are several with the same name. Another question is the type safety guarantees given *after* choosing components.

Our study in [DLE02] is concerned with how Java and C# tackle the second question, i.e., how they guarantee type safety. Dynamic linking in Java and C# are similar: The same linking phases are involved, i.e., loading, verification, offset calculation, and layout determination. Soundness is based on similar ideas: i.e., consistency of the layout and virtual tables, verifying intermediate code, and checking before calculating offsets. Executions which do not throw linking errors give the same result. However, dynamic linking in Java and C# have some differences: The linking phases have different granularity, are combined differently and take place in different order. Linking errors may be detected at different times in Java and C# executions.

We develop a non-deterministic model, which includes the behaviour of Java and C#. We prove preservation properties (which, we consider, were prevalent in the design) prove soundness, and that all executions which do not throw link errors give the same results.

2.3 A monadic multi-stage metalanguage [MF02]

Staging a computation into multiple steps is a well-known optimization technique used in algorithms, which exploits information available in early stages for generating code that will be executed in later stages. Multi-stage programming languages provide constructs for expressing staging in a natural and concise manner, and must allow arbitrary interleaving of *code generation* and *computation*. Multi-stage programming is particularly convenient for defining *generative components* and *mobile applications* (which need to adapt after each move). So far most of the theoretical research on multi-stage programming languages has focused on type systems. The resulting operational semantics are instrumental to a particular type system (thus difficult to relate and compare), and often ignore the subtle interactions between code generation and computational effects.

In [MF02], we provide a deeper understanding of the computational aspects of multi-stage programming in the framework of a metalanguage with computational types $M\tau$ and code types $\langle\tau\rangle$ (computational types classify terms describing computations, while code types classify terms representing other terms), and by describing the operational semantics in terms of two orthogonal ingredients: local simplification (independent from computational effects) and computation (that may cause computational effects). This framework provide a *generic* approach for adding staging to a programming language (described in a monadic style), including a multi-lingual metalanguage, where a variety of monads (programming languages) coexist. In this framework one can also have a fresh look at typing issues.

2.4 Mixin modules and computational effects [AFMZ02]

Mixin modules (or simply mixins) are modules supporting *parameterization*, *cross-module recursion* and *overriding* with *late binding*; these three features altogether make mixin module systems a valuable tool for promoting software reuse and incremental programming. As a consequence, there have been several proposals for extending existing languages with mixins. However, there are several problems to be solved in order to fully and smoothly integrate mixins with all the other features of a real language. For instance, when expressions within mixins can have side-effects, the *evaluation* order of components inside a mixin must be *deterministic* (this could be at odds with cross-module-recursion), it matters *when* and *how many times* computations inside a mixin must be executed. all formalizations defined so far [AZ02, MT00, WV00] do not consider these issues, since they model mixins in purely (functional) settings.

In [AFMZ02] we propose a (foundational) monadic mixin calculus, called CMS_{do}, for studying the interaction between mixins and the store. This calculus should serve as a formal basis both for defining the semantics of imperative programming languages supporting mixins and for allowing equational reasoning. More sprecisely, we combine the pure mixin calculus *CMS* of [AZ02] with a monadic metalanguage equipped with a Haskell-like recursive monadic binding [EL00], and presented in the style of [MF02]. In CMS_{do} the distinction between simplification and computation allows a smooth integration of the *CMS* rules (which are simplifications) with the rules dealing with the imperative features. Since simplification is a congruence, all *CMS* equations (except those related to selection) hold in CMS_{do}.

2.5 MetaKlaim[FMP02, CMSar]

This work is the result of a collaboration with researchers involved in other projects. The goal was to design a calculus addressing the spatial and the temporal aspects of WAN programming, as well as the need to interoperate with potentially untrusted components. MetaKlaim represents one attempt (the first one to our knowledge) at addressing these aspects in a coherent way, and should pave the way to more elaborate proposals (e.g.,incorporating mobile ambients and more accurate type systems).

MetaKlaim is an higher order distributed process calculus equipped with staging mechanisms, which integrates MetaML (an extension of SML for multi-stage programming) and Klaim (a Kernel Language for Agents Interaction and Mobility). It permits the interleaving of meta-programming activities (like assembly and linking of code fragments), dynamic type checking at administrative boundaries and "traditional" computational activities on a wide area network (like remote communication and code mobility). MetaKlaim exploits a powerful type system (including polymorphic types á la system F) to deal with highly parameterized mobile components and to dynamically enforce security policies: types are metadata which are extracted from code at run-time and are used to express trustiness guarantees.

The main conclusions we can draw are: for meta-programming it is important to have staging constructs as well as higher-order (including polymorphism); dynamic type checking imposes a rethinking of some design choices made in [CMSar] for the statically typed language MetaML.

2.6 MoMi [BBV02a, BBV02b]

Flexible communication mechanisms are universally acknowledged as of paramount importance in a distributed mobile code setting. On the one hand, it is important to be "open" enough to accept code for utilities and services that are not exactly as we would have dreamed of (after all if we wanted something perfectly customized we would have written it ourselves, not searched for it on the web). On the other hand, it is also important to be "wise", that is, to be able to single out possibly evil utilities or services from the ones that are harmless.

MoMi (Mobile Mixins) is an "open and wise" coordination language for mobile processes that communicate and exchange object-oriented code in a distributed context. The key idea motivating MoMi is that standard class-based inheritance mechanisms, which are often used to implement distributed systems, do not appear to scale well to a distributed context with mobility. MoMi's approach consists of structuring mobile object-oriented code by using mixin-based inheritance (a mixin is an incomplete class parameterized over a superclass); this is shown to fit into the dynamic and open nature of a mobile code scenario. Mobile code is compiled and typed locally, and can successfully interact with code present on foreign sites only if its type is subtype-compliant with what is expected by the receiving site. Subtyping is therefore used to type check dynamically programs that are communicated across nodes, and thus to guarantee safe, yet flexible, code communication. If the code is successfully accepted, it can interact with the local code in a safe way (i.e., no run-time errors) without requiring any further type checking of the whole code.

References

[AFMZ02] D. Ancona, S. Fagorzi, E. Moggi, and E. Zucca. Mixin modules and computational effects. submitted, 2002.

[ALZ02a] D. Ancona, G. Lagorio, and E. Zucca. A formal framework for Java separate compilation. In B. Magnusson, editor, *ECOOP'02 - European Conference on Object-Oriented Programming*, LNCS, pages 609–635. Springer-Verlag, 2002.

[ALZ02b] D. Ancona, G. Lagorio, and E. Zucca. True separate compilation of Java classes. In *Proc. 4rd Int'l Conf. Principles & Practice Declarative Programming*. ACM Press, 2002.

[AZ99] Davide Ancona and Elena Zucca. A primitive calculus for module systems. In *Proc. Int'l Conf. Principles & Practice Declarative Programming*, volume 1702 of *LNCS*, pages 62–79. Springer-Verlag, 1999.

[AZ02] D. Ancona and E. Zucca. A calculus of module systems. *J. Funct. Programming*, 12(2):91–132, March 2002. Extended version of [AZ99].

[BBV02a] L. Bettini, V. Bono, and B. Venneri. Coordinating mobile object-oriented code. In *Coordination 2002*, volume 2315 of *LNCS*, pages 56–71, 2002.

[BBV02b] L. Bettini, V. Bono, and B. Venneri. MoMi - A Calculus for Mobile Mixins. Submitted, December 2002.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, pages 266–277, 1997.

[CMSar] C. Calcagno, E. Moggi, and T. Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, to appear.

[DLE02] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible models for dynamic linking. Submitted for publication, 2002.

[EL00] Levent Erkok and John Launchbury. Recursive monadic bindings. In *Proc. 2000 Int'l Conf. Functional Programming*, pages 24–35. ACM Press, 2000.

[ESOP00] *Programming Languages & Systems, 9th European Symp. Programming*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[FMP02] G. Ferrari, E. Moggi, and R. Pugliese. MetaKlaim: A type safe multi-stage language for global computing. submitted, 2002.

[MF02] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage. submitted, 2002.

[MT00] Elena Machkasova and Franklyn A. Turbak. A calculus for link-time compilation. In ESOP '00 [ESOP00], pages 260–274.

[WV99] J. B. Wells and René Vestergaard. Confluent equational reasoning for linking with first-class primitive modules (long version). A short version is [WV00]. Full paper, 3 appendices of proofs, August 1999.

[WV00] J. B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In ESOP '00 [ESOP00], pages 412–428. A long version is [WV99].