DOI: https://doi.org/10.53555/nncse.v2i4.453 **Publication URL: http://nnpub.org/index.php/CSE/article/view/453**

SYSTEM PROGRAMMING DRAWBACKS

Abhishek Jain[1]* Manohar Kumar[2], Manohar Kumar[3]
*[1,2,3]*Department Of Computer Science Dronacharya College of Engineering, Gurgaon*
*[1]*abhishek.15004@ggnindia.dronacharya.info*, [2]*manjeet.15056@ggnindia.dronacharya.info*,
[3]*manohar.15057@ggnindia.dronacharya.info*

Corresponding Author: -
Email ID - abhishek.15004@ggnindia.dronacharya.info

Abstract: -

A system programming language usually refers to a programming language used for system programming; such languages are designed for writing system software, which usually requires different development approaches when compared to application software. System software is computer software designed to operate and control the computer hardware, and to provide a platform for running application software. System software includes software categories such as operating systems, utility software, device drivers, compilers, and linkers. In contrast with application languages, system programming languages typically offer more-direct access to the physical hardware of the machine: an archetypical system programming language in this sense was BCPL. System programming languages often lack built in input/output facilities because a system-software project usually develops its own input/output or builds on basic monitor I/O or screen management services facilities. The distinction between languages for system programming and applications programming became blurred with widespread popularity of PL/I, C and Pascal.

Keywords: - *Abstraction, encapsulation, modularity, extensibility, Static type inference*

1) INTRODUCTION

The fundamental problems of a systems-programming task or environment are hard limits on computational resources and a lack of safety protections. Systems programs have to deal with hardware-imposed limitations on their CPU time, registers, memory space, storage space on I/O devices, and I/O bandwidth. They also often have to deal with a lack of the safety features normally given to most programming environments: garbage-collection of memory, validation of memory accesses, synchronization primitives, abstracted access to I/O devices, and transparent multitasking. In fact, the point of systems programming is usually to create such abstractions and program such protections. The properties and requirements noted include:

_ Systems programs operate in constrained memory.
_ Systems programs are strongly driven by bulk I/O performance.
_ Performance and data representation matter.
_ State programming is mandatory.
_ User-managed storage is a requirement.

Languages built for systems programming, therefore, strive to provide the best infrastructure of Programming constructs they can with the thinnest run-time abstractions possible. The ideal is that programmers could write inline assembly in a function's body and still operate easily on data they created in their high-level systems programming language.

2) Precise data representation

Systems programming languages therefore require the ability to precisely represent data structures given in hardware specifications. This requires that the primitive data types available correspond exactly to machine representations, and that complex data types do not, at runtime, contain any data not given in their specification. This has traditionally meant that systems programming languages do not provide very high levels of abstraction at all: C provides primitive data types for signed and unsigned integers, several lengths of floating point numbers, and no more. C's struct types include, possibly, a few bytes of padding to achieve better alignment in cache, but no more; its union and pointer types are untagged. Thanks to these precise correspondences to machine words and registers, C data structures can be used to represent the structures expected by hardware.

3) Unboxed data types for precise representation

All data types in Deca, other than variants, are compiled down to unboxed representations. Parametrically polymorphic functions are type-checked and generalized once, and then instantiated to a separate body of type-specialized code for each assignment of values to type parameters.

The decac compiler therefore performs type specialization and type erasure, leaving no runtime type information. Deca data types thus correspond almost exactly to the LLVM data types they result in. Deca has two different kinds of pointer types: scoped pointers and reference pointers. Scoped pointers come with scope/escape-analysis information attached, and thus allow writing to their referents as though they were variables.

4) Static type inference, despite bit-casting

Deca features strong typing, based on a restriction of the System F<: lambda calculus described by Pierce. The basic calculus was extended with sum types (complete with Pierce's rules for sum-type subtyping), restricted to require record invariance (because record covariance interferes with safe mutability of record fields), and restricted to let-polymorphism (rank-1, predicative polymorphism with the value restriction). Pierce provides preservation and progress proofs for all these features (including their subtyping relations) and for basic System F<: so there exists a provably type-safe core calculus underlying Deca. This is essentially Hindley Milner as it appears in ML, but with subtyping added on. Isolating the unsafety introduced by bit-casts poses a challenge. On the one hand, safety should be machine-checked and language enforced whenever possible. On the other hand, at some point a systems language must take its user's word that their code Is safe. A possible solution is to annotate modules for safety and check the resulting assertions. Only in modules declared unsafe could bit-casts actually be performed; everywhere else, the compiler would forbid them. This would require that programmers document their intent to compromise Deca's safety guarantees, and would encourage the isolation of unsafe code to small, isolated components of the program.

Finally, inference for any specific data variable or function argument can be "turned off" by annotating it with a type.

cast expression:: = CAST < type_form > (expression)

5) Module system and granularity of compilation

Like most modern systems, Deca employs a module system for scoping identifiers. Type, global variable, and top-level function definitions (capable of being polymorphic) are made at the module level. Every source file must consist of one module, which can possibly contain submodules. Unlike some languages, however, Deca considers module definitions one of the forms of definition that can appear inside a module. A module within another module can "see" all symbols defined inside its parent module(s), and this allows compilation of Deca without dealing with mutually-recursive classes, units, or modules. It also allows Deca to subsume the concept of packages into its module system as ordinary modules: a directory filled with Deca source files, considered a "package" in languages such as Java, will in Deca simply constitute a module whose only definitions are those of the modules whose source-files are in the directory.

6) Extensible sum types with minimal records, and encoding object-orientation onto them

Deca provides two mechanisms for extending data types.

The internal mechanism, which users can employ if they please, is open-sum variant types (which can have new cases added to them after their creation). Relatively unusual compilation methods are required to retain separate compilation in the face of open sums. Sum tags must be compiled to the integer values of unique static locations. The size of a variant type's contents must be treated as a weak symbol within the linker, enabling the compiler to write out its final value when compilation of the entire program has been completed and the totality of the variant's cases can be known.

The other mechanism provided by Deca for building extensible data types is classes. A class declaration creates two internal entities: a single variant case corresponding to the declared class (complete with a variant-style constructor function) and an open sum type initially containing only that one case. When one class declaration extends another, the variant case for the new declaration has the member variables of its superclass prepended to its own and is added to the open sum of all its superclasses. This maps class hierarchies onto Deca's open sum types and their subtyping relation. Constructors are specified with a Scala-like syntax of passing expression parameters to the "class" itself, which are then usable in the initializer expressions of member-variable definitions.

7) Modular, symmetric multiple dispatch

Dynamic dispatch of functions has proven itself an essential tool in building extensible programs, but getting its exact style right has proven difficult. Most programs can make do in most situations with the single-receiver dynamic dispatch provided by languages like Java, but languages that only supply single dispatch often require programmers to implement brittle or unwieldly constructs such as the Visitor Pattern, even going so far as to use reflection, to perform dispatch on the dynamic type (or variant case) of more than one method receiver. On the other hand, symmetric multiple dispatch systems must either accept the possibility of an ambiguous dispatch or partially order the set of overriding methods for a generic function into a meet-semilattice. Many practical programs might be rejected by the requirement of forming a meet-semilattice due to meets not existing for some pairs of overridden method cases. Deca escapes this bind using the techniques described by Millstein et al[24]. As they describe their disambiguation restrictions: "We address this problem by restricting EML's function extensibility such that cases declared in modules that do not statically depend upon one another are guaranteed to be disjoint: the cases are not applicable to a common value and hence are not ambiguous."

Deca adds a disjunctive second possibility for the ordering on method cases: if one method M is defined in a module A that statically depends upon (imports) another module B containing a method M', M is more specific than M'. While this does not create the total ordering necessary for completely unambiguous dispatch no matter what overridden method cases exist, it does allow separate, modular type-checking given one more requirement: method cases that are truly incomparable must be disjoint, their type-tuples cannot have a meet which does not contain ?.

In practice, if a programmer wants to write symmetrically-ambiguous multimethod cases, they can specify using module ordering which case is more specific, just as they would order the cases

pattern_list = {one} match_pattern |
{many} pattern_list , match_pattern
pattern_variable = {some} unqualified_identifier | {none} _
match_pattern = {literal} literal_expression |
{variable} [name]:pattern_variable type_annotation? |
{variant} [name]:qualified_identifier (pattern_list?) |
{tuple} [pattern_list] match_case_clause = CASE [pattern]:match_pattern => [body]:expression
match_expression = MATCH (expression) { match_case_clause+ } else_clause? in an ordinary pattern-match.

These requirements result in partial orderings of method cases with unambiguous dispatch. To dispatch a method call, we simply take the runtime type/variant-case tuple of the actual arguments and walk up the partial ordering on method cases from the bottom. When we find a method that can accept the actual arguments, it is the most specific method accepting those arguments and we dispatch to it. Since all methods at equivalent levels of the ordering are disjoint, the ordering itself can even be serialized into a list for more compact representation and faster dispatch.

8) Low-level encodings of high-level features

8.1) Sum types and pattern matching

For sum types, Deca emits an LLVM structure type corresponding to the largest (size in bytes) alternative of the variant, tagged with a variant case-tag. This ensures enough memory is allocated to contain any possible value of the variant type. On the other hand, no runtime type information is included, only the variant case-tag. Deca sum types thus compile down, in the general case, to an LLVM structure with no data not explicitly specified by the programmer. Pattern matching consists of matching against variant case-tags, matching against constant data values (including possibly enumeration values), destructuring records, and simply binding the given data to a new variable.

8.2) Variant representation of lexical closures based on their principal

type signatures Deca represents closure types as a variant with two cases: a function pointer without an environment and a function pointer taking a pointer to an existentially quantified and closure environment. When a function call is performed through a closure, a run-time case dispatch is done to determine what sort of function is being called and to pass it its appropriate environment (or lack thereof). Lambda expressions capture variables by value rather than by

reference, so Deca closures function rather like syntactic sugar for C++ functors, and therefore do not require garbage-collection of stack frames.

8.3) Transparent translation of Deca code into LLVM assembly

In addition to Deca types compiling down to unboxed LLVM data types, Deca language constructs\ compile directly to LLVM assembly code without intervening transformations or virtual machine infrastructure. Code not corresponding directly to the computations specified by the Deca programmer is not emitted.

LLVM, however, does not provide perfect support for all Deca language features, specifically exceptions. LLVM includes limited support for only the C++ model of exceptions and exception handling, and what support it does include is not particularly well-documented. It will therefore take longer to implement exception handling for Deca than other features.

9) Conclusion

Lexing and parsing for decac are implemented using the generator suite SableCC[39], originally chosen so that I could generate a parser from only a grammar and have decac's real internals handle the concrete syntax trees returned by SableCC. Although it does sometimes undergo changes to accomodate changing the language or to better express the intended syntax, the grammar is more-or-less complete and any version committed to the Mercurial repository should result in a buildable parser. While it might, in the future, prove fruitful to translate the Deca grammar into the preferred form of another lexer/parser generator with more mind to specifying the layout of the CST, I do not plan to do so prior to completion of my work on the actual compiler internals.

SableCC is therefore necessary for building decac. Since the decac internals were written in Scala, a working installation of the Scala compiler and runtime is also necessary. I wrote LLVM bindings in Java to work with the Java Virtual Machine version of Scala, and so building or running decac necessitates an installed Java Virtual Machine. If a prospective user or developer of decac does not have the Fast Scala Compiler (fsc) in their installation, they can simply replace "fsc" with "scalac" in the build.xml file found in the main directory of decac. This will rather irritatingly slow down build times, but it builds the same program. Building decac thus also requires the ANT build tool. I selected the most common JVM based build tool. The upside to this requirement is that decac has an automated build process written in an ANT build.xml file rather than an ad-hoc shell script.

Conclusion

Standard I/O is a user-buffering library provided as part of the standard C library. Modulo a few flaws, it is a powerful and very popular solution. Many C programmers, in fact, know nothing but standard I/O. Certainly, for terminal I/O, where line-based buffering is ideal, standard I/O is the only game in town.

Standard I/O—and user buffering in general, for that matter—makes sense when any of these are true. User could conceivably issue many system calls, and you want to minimize the overhead by combining many calls into few. Performance is crucial, and user wants to ensure that all I/O occurs in block-sized chunks on block-aligned boundaries. User's access patterns are character- or line-based, and want interfaces to make such access easy without issuing extraneous system calls. User prefers a higher-level interface to the low-level Linux system calls. The most flexibility, however, exists when you work directly with the Linux system calls. In the next chapter, we will look at advanced forms of I/O and the associated system calls.

References

[1].Eric L. McCorkle. Modern features for systems programming languages. In Proceedings Of the 44th annual Southeast regional conference (ACM-SE 44), pp 691-697, 2006.

[2].Benjamin Pierce. Types and Programming Languages, page 159 and 199. Massachusetts Institute of Technology Press, 2003.

[3].Benjamin Pierce. Types and Programming Languages, page 392. Massachusetts Institute of Technology Press, 2003.

[4].Benjamin Pierce. Types and Programming Languages, pages 187 and 197. Massachusetts Institute of Technology Press, 2003.

[5].Xavier Leroy. Formal verification of a realistic compiler. Communications of the ACM, July 2009.

[6].Gerwin Klein et al. seL4: formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009.

[7].Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pp 282-293, 2002.