C∀, a Study in Evolutionary Design in Programming Languages

by

Rodolfo Gabriel Esteves Jaramillo

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

Abstract

New programming languages appear constantly. Many of them are based on existing ones but differ sufficiently so they are incompatible (e.g., C/C++ and Java). Building on the C∀ language (Ditchfield [47] and Bilson [16]), this thesis continues the C∀ "evolutionary" approach to programming language design based upon the very successful C programming language, preserving its syntax and semantics while extending it with features that considerably enhance its expressiveness. The evolutionary approach allows for the introduction of powerful abstraction mechanisms with minimal disruption to legacy code, truly "making the future safe for the past" [20].

The new features added to C∀ fix existing problems in C and add features that considerably ease the construction and maintenance of C programs. C's declarations and **switch** statement are modified to fix well-known problems. New features in the form of tuples, exceptions and attributes are added to enhance programming and increase robustness. Possible sources of incompatibility are identified and empirically studied.

Acknowledgments

I am much indebted to my supervisor, Dr. Peter A. Buhr for his support, patience, and enthusiasm. Not only did he suggested an interesting thesis topic, but he is a constant source of inspiration in his commitment to teach and in his constant search for knowledge. I am also very grateful to Richard C. Bilson, whose clear thinking often pointed out my mistakes in conception, and whose command of both natural and programming languages frequently solved my problems of expression. I owe Glen Ditchfield many thanks, for his initial design of C∀, which resulted in a beautiful core language. Thanks also go to my readers, Dr. Charles Clarke and Dr. Gordon Cormack, whose suggestions make this a better work.

Waterloo and the Programming Language Group Lab are great places to work at, and I am very happy to have had the opportunity to do so. Many thanks go to the members of this group, especially Egidio, David, Jiongxiong, Ashif and Tom. They forever remind me of the pleasure there is in searching for answers, especially when it implies finding more questions. For a limitless supply of happiness, and a similarly well stocked source of aggravation, I would like to thank Zheng, Kim and Will. God bless you, guys.

Part of this work was funded by the mexican National Council of Science of Technology (Consejo Nacional de Ciencia y Tecnología), whom I gratefully acknowledge.

Dedication

To my parents.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 1962, the historian and philosopher of science, T.S. Kuhn claimed in his influential book *The Structure of Scientific Revolutions* [84] that acquisition of scientific knowledge does not proceed by accretion but rather by revolution: one theory superceding its forerunner by dramatically changing the latter's model of reality.

It would seem that a similar situation occurs to the *theory* of programming, where new tools and techniques supercede and render the old obsolete. Every year a wealth of new, mutually incompatible programming languages make their appearence, each with its own notion of what computation is (on the theoretical side), how particular hardware resources should be presented to the end programmer (on the practical side), and what abstractions need be provided to model problems (on the application side). In other words, each programming language interprets the task of programming in different, incompatible ways. Like Kuhn's theories, every new programming language presents a different view of reality.

However, when it comes to the *craft* of programming, practitioners are much more resistant to adopting new tools. This phenomenon is illustrated by the fact that, among programming languages, the oldest ones are still alive and thriving. When, for whatever reasons, a new language does make its way into the programmers' set of tools, old ideas may need to be expressed in (potentially radically) different notation, if not altogether different notions than before, and require relearning. Moreover, whatever corpus of code is written in prior languages is rendered useless.

This contradiction between the *status quo* of existing programming languages and the complete change associated with new languages is largely unnecessary, since most languages differ from one another in relatively small ways. Languages *evolve* from others languages either by *extension*, adding features the original lacks or provides in inconvenient ways (e.g., original C++ and its C substrate); by *abstraction*, when one or several features of the language are found to be instances of an abstraction mechanism (e.g., the namespace system in C++ is abstracted by packages in Java and assemblies in C#); by *dulcification*, when a commonly used construct in a language is given especial syntax (e.g., a **select** statement in C subsumes a chain of **if-then-else** chain in Algol 60); by *preemption*, when a feature previously left to the programmer's discretion becomes fixed in the language (e.g., different approaches in memory management of C++ and Java); or by *subtraction*, removing problematic or dangerous features in the original language (e.g., static versus dynamic binding in Scheme from –early– Lisp, or Java removing multiple inheritance from C++). Not all these forms of modification (in fact, only the last two) imply the incompatibility of the resulting language with its ancestor. Unfortunately, none implies compatibility.

It is my impression that the art of programming language design lies not so much in providing new language constructs and mechanisms, but in integrating desired features in such a way that they present a convenient notation and consistent model of computation to the end programmer, while allowing for a reasonably fast translation into efficient machine code. This thesis describes the current stage of a project focusing on language evolution in all the forms described above (except preemption), that uses the C programming language as a departure point.

The remainder of the chapter explains the motivation behind C as the substrate and other projects that have undertaken a similar goal. It also gives an overview of the C∀ project and its objectives. Finally, it presents the C∀ language translator architecture, that serves as a testbed for the implementation of the extensions described in later chapters. These extensions include an alternative declaration syntax for C objects (chapter 2), modifications to C's control structures (chapter 3) and *tuples* as a data structuring mechanism, the implications of having such a mechanism weaved into the polymorphic fabric of the language and notational benefits (chapter 4). Chapter 5 describes a mechanism for code annotations that allow the programmer limited access to the structures and information the translation system has generated about the program. In chapter 6, a source code analysis system is described and used to gather evidence as to the relevance of the extensions presented in previous chapters, and estimate the impact of the remaining incompatibilities between C and C∀ on existing code. Finally, chapter 7 presents the conclusions derived from this work and the identified areas of further study. The appendices describe a number of minor additional facilities in the language (e.g., extended numerical literals and composite literals for initialization). Examples of C∀ usage and a summary of incompatibilities with C are also appended.

A remark about the typesetting style used in the thesis. A large number of languages are mentioned in this work for constrasting purposes. The names of some of these languages have no standard spelling, so this work adopts the convention of spelling their names in all capitals if pronounced by enumerating their letters (APL, SQL, etc.), and mixed case otherwise (Fortran, Algol, Cobol). Their defining document cited in the bibliography as their standard, the user manual of the reference implementation, or, failing those, the paper in which the language was first presented. Also, some effort has been made to typeset the code in these languages according to the usual practices in indentation and prettyprinting as applies to the particular language. This might be at times confusing (for example when a token marked as a keyword in one appears as a normal identifier in another), but I thought it preferable to the alternative of a single, uniform, typesetting style.

1.1 C programming language

The C programming language is enormously popular and influential. It has, however a number of troublespots despite several rounds of language revision and standardization. This thesis describes the way a new programming language, C∀, addresses and corrects some of the most insidious of these troublespots, while maintaining almost complete backwards compatibility. This chapter presents some of C's unfortunate design choices and omissions presented in the context of the C programming language evolution. As well, an overview is given for some of the C∀ solutions to these problems.

"Why extend C?" is, oddly enough, a question that is asked both by the C programming language detractors and advocates alike. For the former, C's rule has run its course: the incremental refinement process of the language has gone as far as it can and the burden of backward compatibility is too grievous to carry. It is more productive to start from a clean slate, incorporating the lessons learned in the 30-plus years C has been in use. After all, surely the programming projects now being undertaken are different enough from those programmers had to deal with 30 years ago to warrant a new language. For the latter, C is good as it is, cautiously evolving via the usage-dictated changes introduced by the standardization committee. More radical changes run the risk of not only failing to cure C's maladies but of introducing new ones in the process. This risk is evidenced by the fact that, for every successful spinoff of C, like C++, countless other dialects have been forgotten. Paraphrasing a quip by C.A.R. Hoare, "C is not only a great improvement over its predecessors, but over most of its successors as well" [66].

However, C remains a proven, reliable, precisely defined workhorse, still remarkably healthy and used by many more programmers than C++ or Java. Furthermore, the use of C continues to grow, especially in internationalization projects. One look at the expanding Open Source community provides ample evidence[1]; for example, close to 30% of the projects listed in `freshmeat.com` use C as their development language, as opposed to the 13% that use C++, 13% that use Java, or 25% that use some an scripting language (Perl, Python, Ruby and Tcl combined)[2]. Besides, C is the target language of choice for a number of tools, ranging from parser generators to programming language translators. C has been described as a "universal intermediate representation" [4], or less charitably, a "machine-independent assembly language". C's popularity has made it a frequent means for transmitting knowledge to programmers of every level of expertise. It is common for a programmer these

[1]It has been commented that the Open Source community's marked preference for C is, up to a point, a result of the lack of an available C++ compiler. However, this deficiency is being rectified with the release of `gcc` 3.x.

[2]Sample taken at the end of March 2004. The remainder of the projects hosted by freshmeat are written in other languages.

days to have C as their first (and sometimes only) programming language.

It is in this context that voices of protest start mixing with those singing the praises of C. Its unsuitability as a first language, and as a vehicle for teaching basic programming concepts is consistently pointed out and has been extensively studied [92]. This does not mean that beginners are the only victims to the language's idiosyncrasies; not at all, some of its features and idioms still bite the most seasoned of programmers from time to time [82, 101]. As well, C is often described as deficient when it comes to modern language features. The abstraction mechanisms[1] present in C are very simple (and comparatively not very far from the ones provided by the original Fortran): variables, functions, and preprocessor macros. The omission of more advanced abstraction mechanisms make language support for modern programming practices minimal while making user-defined language extensions hard to write and harder to maintain. This in a world where language extensibility by the user is increasingly high in importance among most programming language developers [119]. Furthermore, programming concepts unfamiliar in the 70s have been explored and have been proven to enhance desirable properties of software, like reliability, maintainability or security.

When a language has stayed in the mainstream as long as C, it is natural to expect a number of attempts to engineer new features while preserving working code and leveraging user expertise. A long string of successors (not as long as Pascal's, but long enough) and a protracted standardization process[2] attest to this. None of these projects has accomplished these goals, but people keep trying and with good reason. Richard Gabriel [57] lists "being similar to existing languages" as one of the characteristics that a programming language should have to increase its chances of success. Given C's popularity, it is not surprising that so many designers have used it as a (at least syntactic) basis for their own creations (e.g., C# and Java).

[1]An *abstraction* or *definitional mechanism* is the set of facilities a programming language provides to give a name to a particular entity (e.g., address of storage, address of an instruction, family of functions, descriptions of types) for later reuse.

[2]While the goals of the standards committee include fixing the language as one of its many objectives, first and foremost, their effort has been to strictly define and then codify best practices, preserving as much working legacy code as possible. Taking this into account, it is understandable that fixing the language did not rank high in their list of priorities.

If the new language in question is not only “similar” but compatible, meaning that all, or a significant portion of the programs written in the base language are valid and correct, the chances of success are conceivably higher. To properly understand how to fix C, it is necessary to describe the C language and its design philosophy, inasmuch as both drive its latest incarnation, the ANSI C99 standard.

In the following section, a short history of C is provided, in the interests of understanding its evolution. Later sections of the chapter describe in some detail the troublespots remaining in the language, and mention which of these are addressed by this work.

1.2 C language evolution

C is a direct descendant of BCPL and, as such, heir to the tradition of procedural languages in the style of Fortran and Algol 60 (figure 1.1). This lineage is still easily noticed: C is small and compactly described, assets well suited to its original domain, system programming. Dennis Ritchie blended these influences with his own interpretations and additions into a design that, as he has pointed out, got things “mostly right the first time” [108]. This basic language is the core around which extensions have been built, and it has come to be known as the “spirit of C”. This term has generated much heated debate when attempts are made to define it precisely; Stroustrup’s version [125] pins it down accurately enough:

1. Keep the built-in operations close to the machine (and efficient).
2. Keep the built-in data types close to the machine (and efficient).
3. No built-in operations on composite objects.
4. Do not do in the language what can be done in a library.
5. The standard library can be written in the language itself.
6. Trust the programmer.

7. The compiler is simple.
8. The run-time support is very simple.
9. In principle, the language is type safe, but not automatically checked (there was `lint` for that).
10. The language is not perfect because practical concerns are taken seriously.

This describes the language that is described in [78]. C acquired more constructs (structure assignment –which to some extent violates point 3 in the list above–, enumerations and **void**), after an unsuccessful attempt by Ken Thompson to rewrite Unix, which is the often-told "trial by fire" of C as a language for programming-in-the-large. This later version of the language is called by Stroustrup "Classic C" [125]. In a process closely overseen by Ritchie, the language was enriched with more types (**long** and **unsigned**), Algol 60-style **union**s, casts, and **struct**s came close to becoming first-class objects (lacking only literals, which would be later added by the C99 standard), and areas of the language were refined, e.g., the relationship between structure pointers and the structures they point to was strengthened.

C spread out in industry and education so quickly and successfully (thanks in no small part to the availability of a portable compiler `pcc`), that the need to strictly define it, was soon apparent. A committee was formed and The ANSI C standard (X3.159-1989) was ratified in December 1989 (although technically it was completed a year earlier). Its charter was aimed more at defining the language with minimal impact to working code than to fix any troublespots. It did fix some troublespots with the introduction of function prototypes, **float** as a data type and the **const** keyword, allowing a modicum of independence from the preprocessor. An ISO C standard followed, but it was in effect equivalent to that accepted by ANSI. In 1992 the ANSI standard was officially withdrawn, as control of the C standard passed from ANSI to ISO, so that now there is a single standard, managed by an international body.

As per these standardization bodies' regulations, starting in 1995 the standard went through a revision process, which yielded the recent C99 standard. A major driving force in this document was the Numerical C Extensions Group, whose intent was to make C a language more suited and appealing to the scientific computation community (boolean and complex types were added, Fortran-style variable-length arrays, more mathematical functions and macros)

Figure 1.1 shows a number of languages and their respective influences, and more importantly, the effect they had on the development and evolution of C. As can be seen, C obtained much of its design and many features from its ancestor language CPL [15]. What the diagram does not show exactly is that the later versions of the language are strictly upward compatible, but lack some backward compatibility [70]. Also missing in the diagram is that the influence of Algol and BCPL transcends a mere particular language design, but represents a *school* of thought advocating that the design of programming languages should be more influenced by the use the programmers put it to than a particular approach to programming thought best by its designers. The C design process was never too detached from this goal, both in implementation and through its user community. (As opposed to Algol 60/68, where the languages were completely designed, and standardized before an implementation was even attempted.) This school of programming language design was called by Richard Gabriel the “Worse is Better” approach [56], and it states that software should start small and evolve according to the needs of its users. Software designed this way, says Gabriel, has a better chance of survival, if perhaps with a diminished aesthetic appeal. In fact, Dennis Ritchie credited this particular design (along with tool availability, right timing and luck) with the success of the language [106].

1.2.1 Success of C

C in its various forms is an enormously successful language. Its portability, terseness, minimal requirements on the run-time system, emphasis on performance and implicit trust in the programmer has made it a favorite among programmers, and (until comparatively recent times) the language of choice where large projects were concerned. Testimonials of this success are the large user-base, huge corpus of code and abundant literature that both focuses on the language or uses it as the medium to introduce programming concepts.

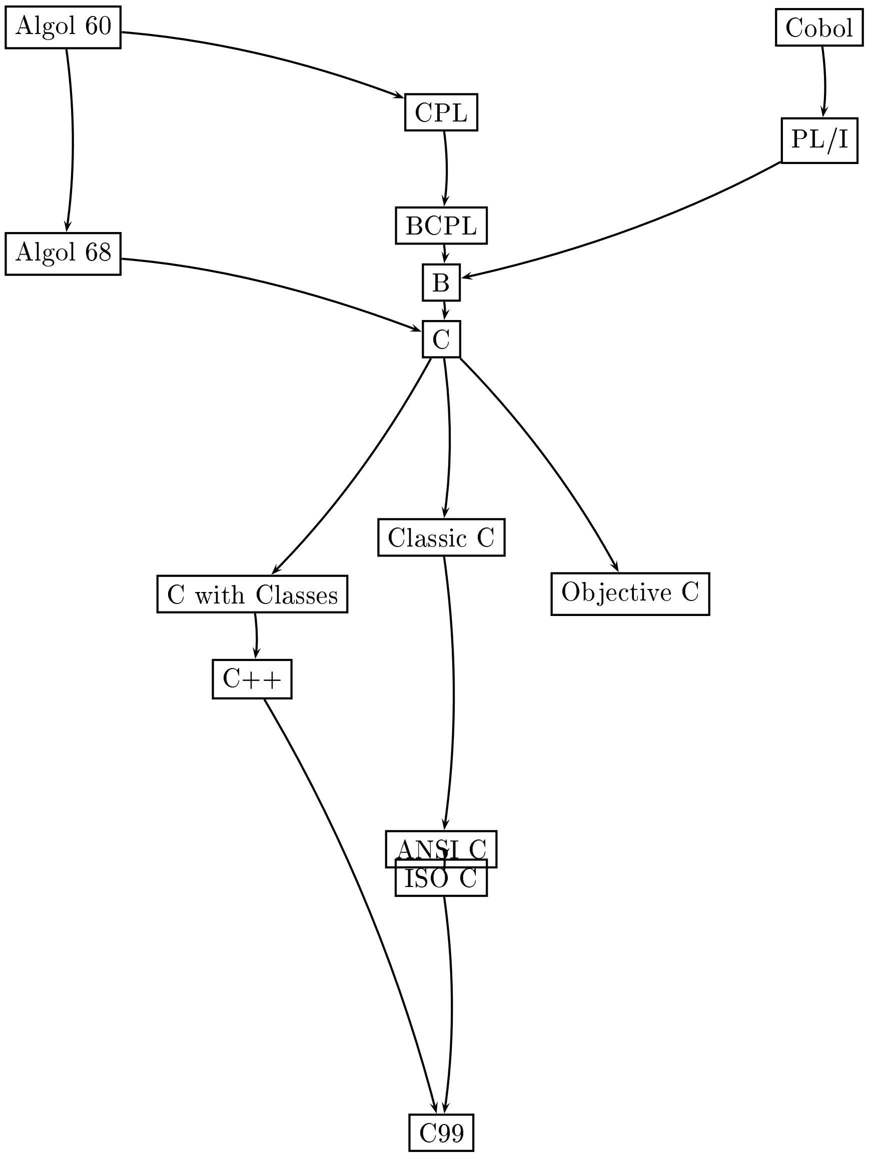

Figure 1.1: The evolution of C and its major influences

Timing was of course an important factor for the widespread acceptance of the language. The success of Unix made C available to hundreds of thousands of people. Conversely, Unix's use of C and its portability to a wide variety of machines was important in the system's success. Fortunately, C and its central library support has always remained in touch with the real environment, which made its transition to platforms other than Unix a relatively easy one. In the words of Ritchie, "[C] was not designed in isolation to prove a point, or to serve as an example, but as a tool to write programs that did useful things; it was always meant to interact with a larger operating system, and was regarded as a tool to build larger tools" [108].

Regardless of the changes mentioned in the previous section, C has remained comparatively stable through a large user-base in a wide variety of environments and with a diversity of compilers. This stability did not prevent a number of dialects from appearing (of particular importance to this document is the one used by the GNU Project's C compiler, gcc), such as the addition of the qualifiers far and near for the Intel segmented architecture.

In contrast to natural languages, the success of a programming language is not measured only by the size of its user-base or amount of code written in it, but along other dimensions: such as expressiveness, reliability, portability, extensibility and support from its environment, among other criteria. The design of C strikes a balance of all these considerations, a balance that is both acceptable and appealing.

Again, citing Ritchie: "C is quirky, flawed, and an enormous success. Although accidents in history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.".

While Ritchie has a modest view of things, another commentator, Richard P. Gabriel was more emphatic: "Right now, the history of programming languages is at an end, and the last programming language is C".

1.2.2 C and C++

Stroustrup intended C++ to be a strict superset of the C language, as C was circa 1980, extending the core language with a stricter type system, some modularity, better data abstraction mechanisms, exception handling, and a number of other features that took care of some of C's more obvious shortcomings [51]. However, the fundamental goal of trying to make high-level paradigms, such as object-oriented programming, coexist and cross-fertilize with other (much lower-level) constructs and programming styles was deemed by some self-defeating. Not only that, but the burden of backwards-compatibility meant that some of the undesirable C features made it into C++, too.

C++ acceptance has been impressive. It is so popular now that the C Standards Committee had to explicitly deny that C++ is the future of C. It is clear now that the languages are following divergent paths. The difference is obvious from the way the languages are evolving: while C is adding more types to the language, C++ is working on its library, using the already-in-place abstraction mechanisms.

1.3 C's shortcomings

As widely accepted and technically remarkable as C is, it has a number of shortcomings in several respects and which need addressing with varying degrees of urgency. These deficiencies are only to be expected, since it is a language that has transcended its originally intended domain of systems programming and has become a general-purpose language.

The most-often (and loudest) voiced criticisms against C are directed towards its lack of readability, which diminishes programmer productivity. With respect to readability, C has a number of inconsistencies, for example, unrelated meanings of the same form (meaning of **static** when it appears in and outside a function, the pointer interpretation of array names, the overloaded **break**, or subtle variations between initializers and assignment expressions), much too low-level pointer semantics (as opposed to, for example, Bliss's or Algol's), and several instances where confusing syntax trips up beginners and experts alike. For example, an awkward type declaration syntax, derived from Algol 68's type composition scheme (although most of Algol's adherents would cringe at the thought).

It has been pointed out that a person conversant in the way computers work and reasonably familiar with computer architecture can immediately make sense of most of C. This quick grasp is even more noticeable in programmers with some experience in compiler writing. It is not out of the question that, at least to some extent, the semantics of the language were motivated by the compiler design, and, what is more, by the compiler technology and programming practices at the time of C's inception [92]. Examples of this abound: the programmer must be aware of what is a compile- and a run-time constant, and C is overly fond of side-effects in expressions (where most programmers trained in other languages, assume expressions denote only values).

The practice of delegating as much as possible to the library (point 4 in the "spirit of C" above) works reasonably well for a number of tasks and subsystems (input/output, for example, where it enhances the portability of the language as a whole), but it seems there are times when the same strategy does not work as well, as is the case with the simulation of several control structures such as exceptions, coroutines, procedure closures, and in particular, concurrency [24], where it has been shown that these abstractions cannot be incorporated into a language via libraries. Furthermore, the restriction imposed by point 5, that language libraries must be written in C, seems overly strict since most modern compilers are able to incorporate procedures written in a different languages, and most modern languages incorporate a "foreign function" interfacing mechanisms.

Another area that needs to be partially resolved within the language, as opposed to a library, is string manipulation. Due to its origins in the systems programming world, C does not offer strong support for character data. The language treats strings like fixed-length arrays of integers (**char**s), with the added guarantee that string literals are terminated with a null character (which therefore can not be contained within a string)[1]. Other than that, C leaves all string processing to

[1]The delimited-string approach in C differs from ancestor BCPL, which uses a length-field.

libraries, and the memory management implied by the manipulation of varying-length strings to the programmer. C's abstraction mechanisms are not powerful enough for the programmer or the library to do a thorough job in either case, since they cannot provide their own copy and assignment operations for specific types. The lack of these operations often results in errors in the use of library functions, which, in the case of the standard string-manipulation and formatted input/output libraries, can lead to pernicious (inadvertent or otherwise) access to the memory regions, in the form of type unsafety or buffer overruns, for example. This problem is detrimental to the overall safety and reliability of the system, particularly in today's computing landscape, where text processing shares some of the dominance that was previously the provenance of numerical computing.

Another problematic area is C's lack of modularity. Modularity is one of the program structuring concepts that, along with data abstraction, was introduced in the 70s, and since then has gained importance as it affects favorably the maintenability of a programming project. C's support for modularity is minimal, which forces programmers to simulate it by distributing their code over files[1] containing preprocessor directives that guide the reassembly of the compilation unit, and then rely on external tools, like Unix `make` for file dependencies and consistency of the project. This approach is a partial solution at best, since more sophisticated devices, like selective imports or exports, data protection in the form of public/private specifications, name collision resolution and all but the more basic data abstraction lie beyond the preprocessor and linker's capabilities. Both the lack of language constructs (such as namespaces, modules and a way to export entities from them while protecting the rest of the code within) and the insufficient preprocessing solution have been widely criticized [93].

This omission is evidence to the fact that execution environments, compiler technology, and coding style (the definition of a good program, the abstractions considered building blocks for good programs, etc.) current at the time (1970s) permeated C's design. Many of these ideas and techniques have since been sur-

[1]C essentialy implements a module with two files, a header containing constants, types and function prototypes, and an implementation file, providing the function bodies, and perhaps **`static`** global variables and internal functions.

passed and the older approaches are now deprecated. This does not mean that code of the highest quality, conforming with one or more of the so-called modern programming paradigms and meeting software engineering requirements in use today can not be written in C. To do so, however, strict adherence to coding conventions is required; but this discipline is not linguistically enforced [93], and therefore, there is no help from the compiler. Hence, even minor deviations from the conventions, either malicious or unsuspecting, make it very difficult for the paradigm to be maintained.

Finally, it is important to mention another source of users' dissatisfaction with the language. C is been used for a wide variety of purposes, some of which fall outside its original sphere of application. For example, as the first programming language, that is, the expository medium through which incipient programmers are taught about basic concepts of Computer Science. As remarkably adaptible as C has proven to be, sometimes even thriving in these unforeseen domains, being a teaching vehicle is not C's strongest suit. C∀ extends C with features that ease this rôle somewhat, such as a more straightforward declaration syntax. However, it is by no means the case that the C∀ project intends to address all such objections in all programming niches.

1.4 C∀ project

The C∀ project charter is twofold:

1. fix C's shortcomings
2. add to C modern programming language features

These changes must be made in such a way that they preserve, from the practical point of view, backwards compatibility with C, and from the philosophical one, "C's spirit": a small and simple language that allows unrestricted access to the underlying architecture and which requires a minimal run-time environment. In this respect, the C∀ project shares the aims of the C99 standardization committee, in that it "attempted to incorporate valuable new ideas without disrupting the

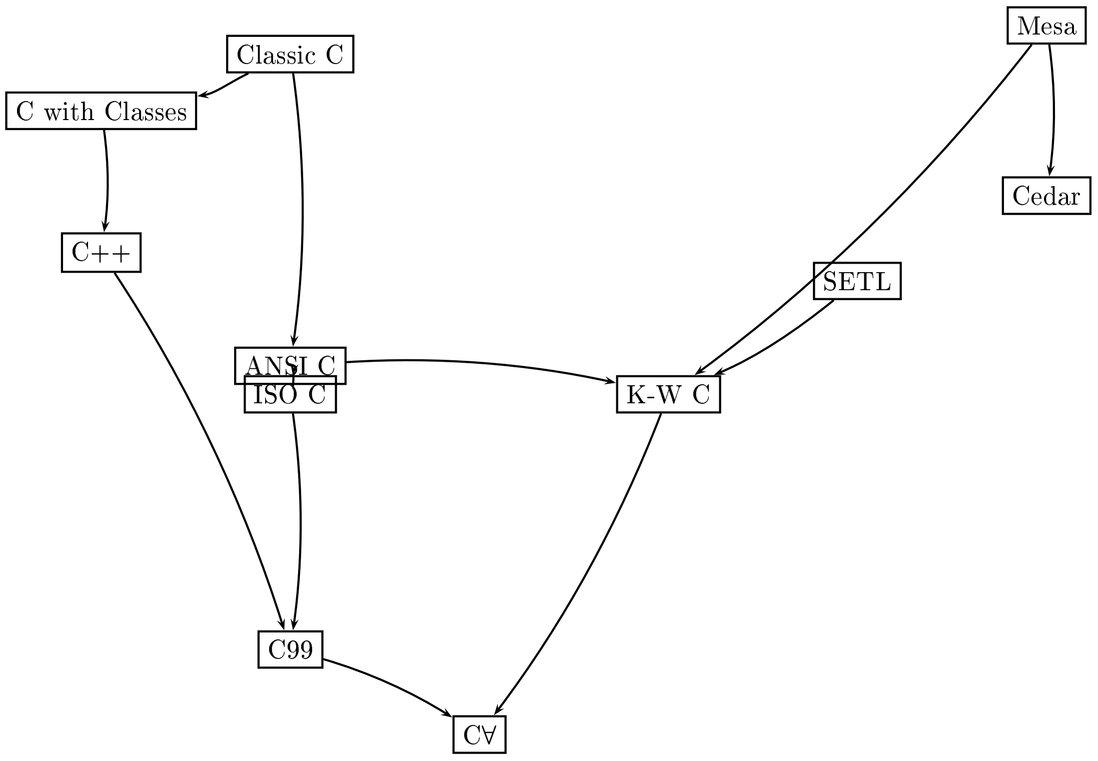

Figure 1.2: Notable influences on C∀

basic structure and fabric of the language. It tried to develop a clear and consistent language without invalidating existing programs" [31]. The difference resides in what sort of "new ideas" these projects incorporate: whereas in Standard C the main focus was on numerical extensions, internationalization and foreign language interfaces, C∀ is not conservative with its additions and modifications, and the introduction of abstraction mechanisms, such as parametric polymorphism, or control structures and patterns, such as exception handling, tuples and functions returning multiple values set the pace for the modifications to the language. Not all these features are new, and their ancestry can be traced in the programming language literature to the 70s (in the case of exceptions, for example). A diagram of the languages that are most influential in the design of C∀ is shown in figure 1.2.

Ever from the inception of C, a great many projects have had the stated purpose of extending that language in a variety of ways and catering to different application domains. However, it is hard to comply with the "spirit of C" as stated above, especially when new abstraction mechanisms are incorporated into the language. Take, for example, C++, whose object-oriented mechanisms for C have been described as an attempt to graft a high-level abstraction mechanism to a much lower-level language, an inherent philosophical clash.

The C∀ approach is different: first it rebuilds the C type system, subsuming C's semantics of expressions, while providing parametric polymorphism and overloading. This type system was designed and described by Glen Ditchfield [47] in 1994. In 2003, Richard Bilson implemented a translator that incorporates Ditchfield's type system, with several extensions [16]. Of relevance to this work is the analysis of expressions involving functions returning multiple values. With this translator as a base, the features described in the present work are introduced.

Contributions of the thesis

The fundamental contribution of this thesis is to make the popular C programming language easier to use and more expressive. While several of the features described in this thesis are similar to those in prior work, their incorporation into C∀'s more complicated type system involved significant differences in design. As well, the mechanisms for implementation of these features from prior work were often not amenable to approaches and techniques used in C∀ and therefore had to be reimplemented.

For example, tuples and their comcomitant operations are derived from those in the forerunner KW-C project [131]. However, adding tuples to C is different from adding them to C∀, given the latter's overloading capabilities. Operations like KW-C's multiple and mass assignment, although similar in form in C∀, have different effects (§4.2). Also, although KW-C tuples imply the notion of functions returning multiple values, their utility is limited by the need to "unpack" the returning values into receiving temporaries, which is cumbersome when functions calls are to be composed. This work removed this limitation, and a novel concept of *tuple designation* was added to allow for the expression of function composition patterns that are not found in other programming languages (§4.2.3). From these extensions, several other natural features were designed and implemented, e.g., named or keyword parameters, default values for arguments, and named return values. Although these latter features are not original, they have proven their usefulness in other programming languages, and their inclusion in C∀ was deemed desirable. I extended Bilson's expression analysis algorithm to encompass these new forms of argument passing and provided a complete translation into Standard C.

C∀ also incorporates other existing C programming constructs subtly modified by this work to enhance usage and prevent misuse. Among these are the variations on C's control structures (Chapter 3), *viz.* loops with multiple-level exits, lists and ranges as **case** guards, and a version of **switch** without default falling through **case**s. I have also added features to C∀, blending ideas from other programming languages. One critically important control structure mechanism missing from C is exception handling. This work added traditional termination exception handling semantics and the more controversial resumption semantics from its forerunner, Cedar. Also, I have designed and implemented a new mechanism for Ada-like attributes that integrates well into the existing syntax and semantics of C∀ (Chapter 5).

While adding all these features to C∀, this work strove to remain within the confines of the "spirit of C" (in particular point 8, above). This constraint markedly separates C∀ from the most visible C variants, Java and C#, and places it within the group of languages that extend C while remaining (mostly) backwards-compatible (some of which are described in the next section). Unfortunately, complete backwards-compatibility is seldom obtainable, and C∀ specifically violates this goal to fix some very questionable features in C. An interesting experiment was designed and performed to assess the effect of all the incompatible changes in C∀ (Chapter 6). The experiment scans a non-trivial body of representative C programs to determine if the incompatible changes introduced in this work have any practical ramifications, i.e., estimate how much real legacy C code would have to be modified to work with C∀. The results of the experiment indicate that very few incompatibilities do occur.

1.4.1 Related work

Narain Gehani is the force behind several dialects of C (C with exceptions, Concurrent C), that extended the language by adding, first exceptions and then concurrency. He made no attempt to fix any existing problems in the language. A similar effort was made by Timothy Budd in Oregon State, who added Icon-style generators to C, and called the blend Cg. Cox's Objective C [38] is an attempt to enriching the C type system with object-oriented constructs. C++ had the same goal, and has already been discussed.

Cyclone [127] places a lot of emphasis on safety. As such, it extends the C runtime environment (it provides "fat pointers" and array subscript checks), introduces region-based (but still manual) memory management and a more strict type system. These extensions prevent errors common in C programs, such as buffer overflows[1]. Its syntactic extensions include exceptions, namespaces, parametric polymorphism and tagged unions.

A number of compilers (commercial and otherwise) extend the language in some ways. `gcc`'s extensions have been mentioned already and are discussed in more detail in the remainder of this thesis. Pike and Thompson's C compiler for Plan 9 [99] provides structure displays (a way to form **struct** expressions dynamically). It also makes anonymous nested **struct**s or **union**s in **struct**s "transparent", so that programmers can refer to members of the inner construct.

It is worth mentioning that the original designer of C was working on a new programming language called Limbo [77] that, although strongly influenced by C, is not backwards-compatible and it changes a number of features, for example, it dispenses with the C-style declaration syntax and adopted Pascal's. It also incorporates a much complicated runtime system, which includes garbage collection, bounds checking and concurrency. This is more a continuous exploration of the programming language design space on their part, and by no means an implicit dismissal of C.

[1]Of course, these problems are not exclusive to programs written in C, but there are features in the language —e.g., pointer arithmetic and unchecked subscripting— that make it easier for the programmer to cause problems, and they go largely unchecked.

1.5 C∀ translator

1.5.1 Black box view

Many descendants of C have been prototyped as preprocessing translators. This particular kind of translator can be described as a compiler front-end: a program that does the lexical, syntactical and semantical analyses and whose output is high-level language program text, in this case C.

This approach has several advantages over building a full-fledged native compiler, such as portability, possibility of interfacing with different optimizers, quicker detection of errors, and several possibilities that the use of one-pass compilers precluded.The strategy is particularly appealing if the output language is C, which has proven to be effective as an “universal intermediate representation” of sorts. Most importantly, a translator of an extended language to the language it extends is significantly simpler to construct.

Of course, this comes at the cost of a slight degradation in performance when compared to a native compiler. Furthermore, if not enough information is passed to the output program, the capability of symbolically debugging the original program is also diminished.

However, there are language features that simply cannot translate directly to the target language, or that translate only partially. For example, exception handling had to be done in the most straightforward of ways since the translator has no access to lower level information, in particular the addresses in memory a particular block of code occupies when translated to machine language.

1.5.2 Functional description

C was originally designed with a bottom-up compilation model: no intermediate representation is necessary and the code can be generated as soon as possible [1]. The

[1] As opposed to other languages, for example Ada, where the assumed compilation model depends on a tree representation.

more elaborate mechanisms introduced in C∀ preclude this mode of operation, and a staged translation is required. The parser constructs an intermediate representation that is later traversed in various ways until finally the code is generated.

Two intermediate representations of the program are built. The first is closely tied to the grammar of the language, and more amenable to reflect future modifications. To some extent, it also reflects the development tools used (`flex`, `bison`, etc.) and the at times awkward interaction with one another (e.g., C++ STL and `bison`).

Once the building of this representation is completed, a second representation, modelling the semantics of the language more closely (in effect, an Abstract Syntax Tree), is constructed. Once this second structure is in place, the translation process is carried out by several passes that rewrite sections of the AST. Some of these rewriting passes are described in much greater detail in the remainder of this document.

After all these rewriting passes have been completed, what remains (for a valid C∀ program) is the AST of a C program, which can then be mapped to a string that is the output of the translator. A driver takes this output and, then feeds it to the compiler, along with a number of compiler flags.

Chapter 2

Declaration Syntax

Ever since Algol introduced the concept of declaration of objects in a program[1], and that these declarations are decorated by the type of the object, a sublanguage for type description had to be incorporated into programming languages. In most languages, starting with Algol W and Algol 68, type specifications are built incrementally from a collection of *primitive* or *basic* types by the (repeated) application of type constructors.

In C, declarations are composed of two parts, the base type and a *declarator*, which includes the name of the entity. If an object is not of a built-in type, a description of the type is provided in its declaration, and this description is composed of basic types, type synonyms (introduced via **typedef**) and constructed types. Type constructors include a pointer constructor (*), that takes a type as argument; an array([]) constructor that takes a type and possibly a dimension specification; a function constructor that takes a return type and a list of parameter types; the record constructor (**struct**) and the undiscriminated disjoint union constructor (**union**), that take list of types as *members*, together with a name for each of them; or enumeration constructors (**enum**), that takes a list of *enumeration constants.* A type specification might be further adorned with *qualifiers* (in C99 **const**, **restrict** and **volatile**). Finally, a declaration can be further adorned with one of six storage

[1]The C standard [6] uses the word "object" to denote a region of memory containing a value, as opposed to other interpretations in the context of different programming paradigms.

classes: **auto**, **extern**, **register**, **static**, **inline** and **fortran**. Any combination is allowed as long as it makes some sense.

Throughout the course of its evolution, C's declaration syntax has changed somewhat. In its first incarnation, and all the way through the version described in the first edition of K&R, C did not require the specification of the base type of an object, nor did it require specifications for the return type or number of arguments in a function. Where applicable, these objects' type was considered to be **int** by default:

```
x;        // same as int x
*x;       // same as int *x;
foo(p);   // int foo( int p );
foo(x,y)  // K&R old-style function declaration
  int x,y {
    // same as int foo(int x, int y)
}
```

Pointers were declared exclusively using the array constructor rather than a star [107]:

```
int ip[]; // instead of int *ip;
```

a notation that survives to this day in a vestigial form in argument declarations. C89 introduced type specifiers (**const** and **volatile**), function prototypes (that allowed programmers to specify the argument types in a function declaration, rather than in a function definition) and some additional qualifiers (e.g., **unsigned**). C99 deprecated the interpretation of missing base types meaning **int**, and incorporated more primitive types and the **restrict** qualifier.

Unfortunately, these alterations have done little or nothing to change the fact that the declaration syntax of C is one of the most often voiced complaints about the language.

2.1 C Declaration Syntax

C is a direct descendant of B, B of BCPL, and BCPL of CPL (figure 1.1). CPL has a design so ambitious that it was not entirely implemented. BCPL (Basic CPL) restricted CPL in several ways, one of which was to substitute the diversity of types with a single one, the word. B further scaled down BCPL, specializing it for systems programming. Because of BCPL's word-based system was insufficient for the purposes of Thompson and Ritchie, C adopted parts of CPL's type system, albeit in a different guise. In particular, C's declaration syntax is original and completely different from CPL's (which follows Algol's).

C's choice of syntax for declarations is based on the idea that the declaration of an entity should look like the use of that entity in an expression. To this effect, tokens denoting operators are recycled in a declaration context to adorn base types and entities. Although this design complicates the parsing of the language[1], the symmetry between declaration and use was considered by the inventors worth the trouble.

At a first glance, this schema is both elegant and effective, consider:

Description	Declaration	Use
int variable	**int** x	x
pointer to an **int**	**int** *x	*x /* *dereference* */
array of 10 **int**s	**int** x[10]	x[5] /**subscript* */
function taking and returning an **int**	**int** foo(**int**)	foo(x)
function taking an **int** and returning a pointer to an **int**	**int** *foo(**int**)	*foo(x) /* *dereference* */

[1]When parsing bottom-up, it is unclear, upon encountering a type constructor token, whether the correct parse tree results in an expression or a declaration, and more context information is needed.

However, this approach breaks down as more complicated objects, combining arrays, pointers and functions, are used:

Description	Declaration	Use
array of ten pointers to **int**	**int** *i[10]	*i[5] * /* subscript binds tighter */*
pointer to array of ten **int**	**int** (*pi)[10]	(*pi)[5] */* deref and then subscript */*
pointer to a function returning an **int**	**int** (*pf)()	(*pf)()
array of pointers to functions returning **int**	**int** (*apf[10])()	(*apf[5])()

When both post- and prefix type constructors are used, a declaration specification is layered (like an onion) around the object name, and is read from the inside out. However, this rule of thumb does not begin to clarify declarations like the ones presented above.

The use of type synonyms introduced via **typedef** can ameliorate to some extent the need for complicated type declarations by allowing the incremental construction of the desired type ([39]):

```
typedef char fch();     // function returning a char
typedef fch *pfch;      // pointer to function returning char
typedef pfch *apfch[10]; // array of pointer to function returning char
typedef apfch *papfch; // pointer to an array of pointers to functions...
```

This is, at best, only a partial solution to the problem, especially when type qualifiers and storage classes are brought into play. Type qualifiers and storage classes can appear in many orders. The next lines all describe the same object:

```
const int volatile x;
int volatile const x;
const volatile int x;
```

In fact, any of the seven possible versions of a qualified type is valid, although objects declared with different versions are incompatible with each other and with the unqualified type. Type qualification is only relevant in a lvalue context, that is, when an object with qualified type appears on the left side of an assignment or in the parameter list of a function prototype. In most other cases, a qualification in any other position is dropped. For example, the following is valid:

```
void bar( const int x );

void foo() {
  volatile int vcpi;
  bar(vcpi);
}
```

Syntactically, the type qualifier **const** presents particular difficulties when coupled with pointer variables. Since a pointer declaration describes what type the identifier is supposed to point to, it actually refers to two entities: the identifier and the entity it points at, either of which can be used in an lvalue context.

```
int const *pi;
const int *pi;
int * const pi;
const int * const pi;
int const * const pi;
```

The first two lines describe the same object, a pointer that points to a **const int**, that is, the pointee is not to change (in other words, any dereference of the pointer in a lvalue position is invalid). The next line describes a pointer that cannot point to a different target (direct assignment and modifying pointer arithmetic, therefore, is forbidden, but an assignment to the dereference is allowed). Finally, the last two lines, describe a pointer that cannot change its target nor can it assign to the object it points to.

It is debatable how much the "declaration mimics usage" policy is to blame for how error prone the notation turned out to be. Another factor that compounded the confusion is the existence of prefix and postfix type constructors, that can be combined in various ways. This has been deplored by Dennis Ritchie himself [1], who recognized that several syntactic and lexical mechanisms of CPL, including procedure and data declarations, are more elegant and regular than those in C.

Another problem comes from the possibility of declaring more than one object in the same declaration statement, but only the base type is distributed across all variables:

```
int* x, y;  // x is a pointer, y is an int
int *x, *y; // two pointers to int
```

Beginner programmers, and even experienced ones with some familiarity with Pascal descendants are often surprised to discover the meaning of declarations like the above, especially since errors are not issued by the compiler until inappropriate application of an operator, potentially far removed from the declaration itself.

In conclusion, C declaration syntax is fraught with complications, which still trip even experienced programmers[2]. These complications are so fundamental that

[1]Ritchie has suggested that the declaration syntax would have worked much better had the indirection operator been postfix rather than prefix. In Pascal, where declaration syntax is not consistent with usage, pointers look like:

```
var iptr : ^integer;
⋮
new(iptr);
iptr ^ := 10;
writeln('the value is ', iptr^ );
dispose(iptr);
```

[2]Even specialized tools, like `cdecl`, that translate a C declaration into English and viceversa have been developed.

successive revisions of the language have not solved the problem. A more radical change thereto is necessary.

2.2 C∀ Declarations

Till [131] constructed the first implementation of an extended declaration syntax; however, this implementation required several additional keywords to disambiguate the grammar. Buhr et al [26] constructed the second implementation but still required one additional keyword for disambiguation. In C∀, I eliminated all superfluous keywords previously needed, and extended the C∀ declaration grammar to incorporate the new declaration syntax among the other polymorphic extensions in declarations.

C∀ provides simpler type, variable, and function declarations. All the tokens denoting type constructors and their meaning are retained, but they are all prefix and right-associative. This greatly simplifies complicated declarations:

Description	C	C∀
int variable	**int** x	**int** x
pointer to an **int**	**int** *x	* **int** x
array of 10 **int**s	**int** x[10]	[10] **int** x
function taking and returning an **int**	**int** foo(**int**)	[**int**] foo(**int**)
function taking an **int** and returning a pointer	**int** *foo(**int**)	[**int** *]foo(**int**)
array of ten pointers to **int**	**int** *x[10]	[10] * **int** x
pointer to array of ten **int**	**int** (*pi)[10]	* [10] pi
pointer to a function returning an **int**	**int** (*pf)()	* [**int**]() pf
array of pointers to functions returning **int**	**int** (*apf[10])()	[10] *[**int**]() apf
array of pointers to functions	**int** (*(*papf)[10])()	* [10] *[**int**]() papf

Even the most complicated declarations can now be read left to right without complex binding rules to remember, and without having to resort to auxiliary **typedef**s.

Qualifiers and storage classes are placed to the left of the base type, but are otherwise used in the normal way with the new declarations:

Description	C	C∀
const pointer to **const int**	**int const** * **const** x	**const** * **const int** x
const pointer to array of 10 **const int**	**const int** (* **const** pai)[10]	**const** * [10] **const int** pai
extern array of 10 **int**s	**int extern** ai[10]	**extern** [10] **int** ai;
static pointer to **const int**	**const int static** * pi	**static** * **const int** pi

When declaring multiple entities, the entire type specification is distributed across all variables in the declaration list:

```
* int x, y; // two pointers to int
* [10] * int x1, y1; // two pointers to array of pointers to int
```

Note that in C∀ a function return type is enclosed in square brackets, and that an empty parameter list denotes a function taking no parameters (as in C++) whereas in C the same notation means an unspecified number of parameters.

```
[] g();        // void function taking no arguments
[int] f();     // function returning int taking no arguments
[char,int] f();   // function returning a tuple (cfr. chapter 4)
```

Finally, declaration qualifiers and storage classes are only allowed to appear at the start of a C∀ routine declaration[1]:

```
extern [ int x ] g( int y ) {}
```

Different styles of type specification can be used in separate declaration statements (but not intermixed in the same declaration), even in the same block of code. Likewise, either style can be used in any context that requires a type specification, for example, a cast, **sizeof**, or **typeof** contexts. The only case where intermixing of declaration styles in the same declaration is allowed is declaration of function return and parameter lists of C∀-style functions:

```
[int (*x)[10] ] f( int (*y) [10] ); // C-style return and parameter declarations
[* [10] int x] g( * [10] int y );    // C∀-style return and parameter declarations
```

This exception allows backwards compatibility with old-style macros generating C declarations, e.g.:

```
#define ptoa( n, d ) int (*n)[d]
[ ptoa(x,10) ] f( ptoa(y,10) );
```

In general, intermixing declarations styles is neither recommended nor supported, as this practice tends to compromise clarity. It is hoped programmers (especially new ones) will prefer C∀'s declaration style to C's.

C∀-style declarations are rewritten during parsing to their equivalent C forms, which means that the use of either form of declarations does not affect performance in any way.

[1]C99 has adopted the same rule and deprecated alternate usage. It is, of course, up to time to determine how effective this policy will be, but due to the extant legacy code, it is safe to say that it is hopeless.

2.3 Related work

C's syntax for declarations has been the constant source of complaints and a significant number of reengineering efforts have gone into making the sublanguage more readable, for example by Anderson [10] and Sethi[112]. Sethi's proposal is notable because he made the indirection operator postfix. The style of declarations this modified syntax allows is much clearer than C's (in Sethi's proposal, the indirection operator, denoted in C by * is written ^):

```
char (* (*x[3])())[5]; // C
char x[3]^()^[5];       // Sethi
[3] * [ * [5] char ] () x; // C∀
```

Sethi's syntax is clearly more readable than C's, and probably just as clear as C∀'s. Interestingly, it still allows a programmer to reuse the "char" at the beginning of the declaration for subsequent declarations, in the form:

```
char x[3]^()^[5], y^[5], z;
```

However, the distance between the type specifier and the second and third variables is large, and not conducive to clarity. C∀ does not suffer from this problem.

Several other attempts have been made to fix C's declaration syntax, which some subsequent languages inherited. For example, Werther et al. [135] suggest changing the C++ syntax declaration, to not only incorporate Pascal's ^ pointer constructor, but also ML's function type constructor –>. C∀'s solution requires less syntactic innovations, while attaining the same clarity of expression.

Chapter 3

Control Structures

Like most imperative languages, C includes the traditional control structures of unconditional jumps, conditional branching and selection, and looping, largely derived from those in BCPL and Algol. C∀ follows suit, but it introduces some variations in the interests of program readability and maintainability. This chapter describes how I augmented C∀'s control flow by fixing and extending existing C constructs and adding new ones. Further changes in control-transferring mechanisms, specifically function calling, are described in following chapters.

3.1 Multi-level exits

The structured programming school introduced the notion that certain code structures, e.g. loops and subroutines, should have only one entry and one exit. While the one-entry restriction seems logical and has remained largely uncontested, experience suggests that having multiple exit points is useful and sometimes necessary. In fact, many programmers simulate multiple exit points by testing a multitude of logical flags, resulting in code that is so difficult to read and maintain that these use of flags is often considered the data equivalent to the unstructured use of the **goto** statement.

To alleviate this problem, language designers have provided facilities to exit loops and functions at different points. In C, these facilities take the form of the

```
register t;
t := head;
while (t := ..t) _ 0 do
  while (t := ..t) _ 0 do
    if .(.t + 1) _ 0 then exitloop[2] .t
```

(a) Numbered exit

```
register t;
t := head;
l: while (t := ..t) _ 0 do
  while (t := ..t) _ 0 do
    if .(.t + 1) _ 0 then leave l with .t
```

(b) Labeled exit

Figure 3.1: Multilevel exit in Bliss

statements **break** and **continue** for loops, and **return** for functions. **continue** is valid only inside a loop and when encountered causes the current loop iteration to terminate and control flow transfers to the beginning of the closest enclosing loop to start a new iteration. **break** can be used in a similar fashion, with the sole difference that all loop iterations are terminated and control flow transfers past the end of the closest enclosing loop. Buhr [23] calls this construct a multi-exit loop.

However, a C multi-exit loop is limited in the sense that it is restricted to the closest enclosing loop. A generalization of this capability is a *static multi-level exit*, which allows transfering out of a multiple nested construct to a statically determined location. Bliss [140], a systems programming language designed at CMU in the 70s and no longer in use, contains the equivalent to C's unqualified **break** (of the form **exitloop**[1]), as well as a multi-level exit statement of the form **exitloop**[n], where the parameter n indicates the number of control structures to be exited from. This construct is understandably hard to maintain, since adding or removing nested control structures required updating all the occurrences of n. Later versions of the language allowed the loops to be labeled (see Figure 3.1). Peterson et al. [97] discuss the capabilities of similar constructs *in extenso*.

Ada [48] combined both forms by allowing exit statements to explicitly specify the enclosing control structure out of which control is to transfer; if the label is missing, the innermost control structure body is assumed. Several programming languages (e.g., Java and Perl) adopted this construct. Examples of this device can

[1]**exitloop** takes an optional return value, since control structures in Bliss are expressions, rather than statements.

```
OUTER: foreach $i (@where) {
    foreach $j (@{$i}) {
      # single-level break
      last if $j == −1;
      last OUTER if $j == $what;
    }
}
```

(a) Perl

```
Outer:
for I in Where'Range loop
   for J in Where(I)'Range loop
     −− single-level break
     exit when Where(I)(J) = −1;
     exit Outer when Where(I)(J) = What;
   end loop;
end loop;
```

(b) Ada

Figure 3.2: Multilevel break

be seen in figure 3.2. As well, languages like Java and C# allow for general blocks (i.e., those that are not the body of a looping construct) to be labeled and referred to by multi-level exit statements.

C∀ adopts C's looping constructs unchanged, and extends them to the more general scheme of Java and C#. This is, labels identifying a *block* can be used in a **break** statement to transfer control to the instruction immediately following the end of the block. Loop statements can also be labeled and the behaviour is equivalent to the loop block being labeled. Also, **continue** statements within a loop can refer to these labels to immediately proceed to the next iteration.

In effect, all of these constructs are a restricted forms of **goto**. They allow only branching to the beginning and the end of enclosing control structures, and therefore cannot be used to create cycles in the control graph. In this way, construction of loops is restricted to the language loop statements, and a legitimate use of **goto** [81] is given a new syntactic form. The C∀ translator makes use of the lower-level **goto** to implement these extended control structures, as can be seen in figure 3.3. Notice that a **continue** or **break** referencing the closest enclosing loop or **switch** is not transformed into a **goto**.

```
Block: {
    Loop1: for ( i = 0; i < 10; i += 1 ) {
      Loop2: for ( i = 0; i < 10; i += 1 ) {
        Switch: switch ( i ) {
          case 1:
            if ( i < 5 ) {
              continue Loop1;
              continue Loop2;
              continue;
              break Block;
              break Loop1;
              break Loop2;
              break Switch;
            } // if
        } // switch
      } // for
    } // for
  } // block
```

```
 __L0__: {
  __L3__: for (i=0;i<10;i+=1) {
    __L2__: for (i=0;i<10;i+=1) {
      __L1__: switch (i) {
        case 1:
          if ( i<5 ) {
              goto __L5__;
              continue;
              continue;
              goto __L6__;
              goto __L7__;
              break;
              goto __L4__;
            } // if

          ;
        __L4__: break;
      } // switch

    } // for

    __L5__: /* null statement */ ;
  } // for

  __L7__: /* null statement */ ;
} // block

__L6__: /* null statement */ ;
```

Figure 3.3: Multilevel exit in C∀(left) and C translation (right)

3.2 Selection statements

An often-encountered need during programming is to pick out one course of action among several (excluding all others). Characteristically, programming language designers went about providing for this necessity by successive approximations: first there was the two-way selection statement or expression, almost universally taking the form of an **if**. This simple binary branching can be scaled to an arbitrary number of cases/actions by nesting or cascading **if-then-else** statements. When cascaded, the number of **if**-guards or test conditions often correspond to mutually exclusive (and presumably non-overlapping) alternatives. Several proposals to provide some syntactic sugaring to such a programming artifact have been advanced.

If the test is on the same variable or expression (and this is of an ordinal type), writing the condition repeatedly is tedious, and often introduces mistakes. C.A.R. Hoare[1] introduced the multiway branching statement in Algol, called a **case** statement, which is considered a better solution to the above problem, as it is more readable and less prone to mistyping of the condition.

Most languages offer a form of the multi-way selection statement (**switch** in the C-family, or **case** in Pascal's descendents, for example), with some variations, mostly in the kind of expressions that can be used to label the cases (lists of values, ranges, even full-fledged predicates). Different types of guarded expressions have also been extensively experimented with (Unix shells, to cite an extreme example, offer a combination of a string test expression and regular expression patterns as case labels).

The form this statement has taken in C has been widely criticized, and it is often listed among the most annoying and dangerous characteristics of the language. Originally inherited from BCPL [106], it took a somewhat idiosyncratic bent when "falling through" **case** clauses was made the default behaviour. In most other languages, the execution of the code associated with a particular **case** clause precludes the execution of any other clause in the same **switch** statement. In par-

[1]Hoare said of this construct "This [the **case** statement] was my first programming language invention, of which I am still most proud, since it appears to bear no trace of compensating disadvantage" [66].

ticular, during the execution of a **case** clause, the presence of the following **case** clause indicates the completion of the action triggered by the alternative and the transfer of control to the next statement after the **switch** statement. In C, the programmer has to explicitly indicate this transfer by inserting a **break** statement to exit the **switch** statement. The rationale behind this design is that it makes up for the lack of ranges and more complex expressions in **case** labels, but it is a decision that still baffles beginners (especially if they have been exposed to other languages with different semantics).

The overloaded use of the name **break** (used both to exit **switch** and loop statements) does not help matters either. In the interest of minimalism (or probably as a result of his using an outdated BCPL manual [106]) Ritchie decided to reuse the keyword **break** rather than introduce another (BCPL settled for the keyword **endcase**). This decision has proven to be unfortunate, as it leads to errors, an extreme example of which is the one that took down AT&T's whole long-distance network [3], which has been attributed to a bug caused by a **break** incorrectly associated with a **switch** when the programmer's intent was to exit the enclosing looping statement.

Interestingly, the C-inspired languages, most notably Java, preserve falling through **case**s as the default behaviour, with experienced C programmers more in mind than beginners (some C programmers even design the **case** labels in their code to *rely* on fall-through). Microsoft's C#, one of the latest additions to this family of languages, departs from this tradition by assuming fall-through when faced with empty **case**s, but syntactically requiring a **break** at the end of all non-empty **case**, and, should fall-through be required, forcing the programmer to make it explicit via **goto**s. C#'s attempt, commendable as it may be, will probably throw off old-school programmers by placing additional constraints on a familiar structure.

The C∀ solution goes halfway, keeping the traditional **switch** construct (almost) unchanged, but also providing labeled **break**s (§3.1), which help prevent errors like the one described above, by stating explicitly *what* construct the **break** is intended to exit, and by introducing a new control statement, called **choose**, described below.

As much as the preservation of the **switch** semantics was a design goal (for backwards compatibility), some changes were introduced for the sake of uniformity with other concepts in the language. Since these changes are incompatible with C, the translator might signal working C code as invalid C∀. These changes are the main sources of incompatibilities between C and C∀ programs, and, along with the clash of identifiers with new keywords, the most likely to create problems with legacy code. In practice, however, real code seldom makes use of the C constructs that C∀ changes, and therefore, the impact of the incompatibilities is minimal (for a systematic study of these issues, refer to chapter 6).

The major difference between C∀'s **switch/case** and C's is that **case** clauses are not allowed anywhere but in the (first-level) body of a **switch** statement. This restriction is geared to prevent jumping *into* a control structure, with the possibility of bypassing invariant-checking code, variable declaration or initialization and other crucially important statements. The ability to 'interweave' a **switch** with other control structures, along with the default fall-through from **case**s, allows for code like the classic *Duff's device* (figure 3.4). Duff's device has been called "the most dramatic use yet seen of fall through in C". It was discovered by Tom Duff, while unrolling loops in the interests of performance and rewriting the unrolled versions by interlacing the structures of a **switch** and a loop (resulting in a so-called "unnatural loop" [94]). While Duff himself could not place his device as an argument for or against fall-through [118], C∀ designers had no such qualms and set Duff's device, for all its potential efficiency gain and ingenuity, squarely in the curiosity bin.

A minor difference between C and C∀'s **switch** statements concerns code placed between the **switch** and the first **case** label. Code in that position is syntactically valid in both languages, but it is handled differently. While C does allow declarations to have a scope that includes all the **case** clauses, it does not guarantee that initialization of these entities is performed. Other than declarations, code in this position is considered "unreachable" and can only be executed if it is accessed through a **goto** statement. This behaviour is confusing and inconsistent with all other declaration contexts in the language. Therefore, C∀ adopts alternative semantics: initialization of these declarations is guaranteed. To achieve this semantics, the translator generates an extra block around the **switch** and hoists

```
register n = (count + 7) / 8;   /* count > 0 assumed */

switch (count % 8)
{
case 0:        do { *to = *from++;
case 7:             *to = *from++;
case 6:             *to = *from++;
case 5:             *to = *from++;
case 4:             *to = *from++;
case 3:             *to = *from++;
case 2:             *to = *from++;
case 1:             *to = *from++;
                  } while (--n > 0);
}
```

Figure 3.4: Duff's device

initial declarations to the front of that block. This transformation is illustrated in figure 3.5.

3.2.1 Case labels

Historically, the **switch** statement descends from the **switch** construct of Algol 60, which is a specialization of Fortran's computed **goto**s. C's **switch** differs from Algol's in a number of respects, most notably in the fact that ranges are not allowed as **case** labels (although they were considered for inclusion in the process of developing the ANSI C89 standard [9]).

Part of the rationale behind the default fall-through between **case**s is that the same code can apply to a variety of options, which permits code reuse. It also gives rise to a number of idioms that (partly) make up for the limited capability of the

```
switch(i) {
  int i = 4;
case 0:
  i = 17;
  /* fall through */
default:
  printf("%d\n", i);
}
```

(a) C∀ version

```
{
int i = 4;
switch(i) {
case 0:
  i = 17;
default:
  printf("%d\n", i);
}
}
```

(b) C translation

Figure 3.5: Declaration hoisting in selection statements

value labels, in particular, for lists of unrelated values, for example:

```
switch ( i ) {
case 1:
case 2:
case 8:
case 22:
case 43:
  do_something();
}
```

Although this is a well-established idiom, it often falls short of its aim. In particular, consider the code fragment:

```
switch ( i ) {
case 1:
case 2:
case 3:
  code_range_1to3();
  break;
case 50:
case 51:
case 52:
  code_range_50to52();
  break;
}
```

and similar instances when consecutive values have to be manually enumerated, a task that is both tedious and error-prone. C∀'s designers have determined that the introduction of more elaborate case labels is necessary. A programmer can specify a range of consecutive values in a case label by using either gcc-style (start ... end) or C∀ start~end syntax[1], where start and end have to be of comparable integral types, for which an incrementing or decrementing sequence can be inferred (i.e., they may be integrals or enumeration types in ascending or descending order). Failure to comply with this requirement results in the compilation process issuing a type-check error.

Ranges are implemented by expansion to the corresponding falling-through case statements, thereby introducing no performance degradation compared to standard C. A more direct translation into gcc ranges could conceivably be compiled into a more efficient representation. An illustration of the kind of transformation currently implemented in the system is presented in figure 3.6.

[1]C∀ adopted a different syntax than the gcc's ellipsis to denote a range, because its specification, `1␣...␣10`, requires spaces around the ellipsis, otherwise it is tokenized as "1.", causing a syntax error.

```
typedef enum
  { ONE, TWO, THREE } options_t;

⋮
  switch ( i ) {
  case ONE ... THREE:
    printf("In enumeration\n");
    break;
  case 3 ... 5:
    printf("Between 3 and 5\n");
    break;
  case 6 ~ 10:
    printf("Between 6 and 10\n");
    break;
  default:
    printf("Too high (%d)!\n", i);
    break;
  }
```

```
enum __anonymous0
{
    __ONE__C13e__anonymous0,
    __TWO__C13e__anonymous0,
    __THREE__C13e__anonymous0,
};

⋮
  switch (__i__i) {
        case __ONE__C13e__anonymous0:
        case __TWO__C13e__anonymous0:
        case __THREE__C13e__anonymous0:
           printf("In enumeration\n");
            break;;
        case 3:
        case 4:
        case 5:
            printf("Between 3 and 5\n");
            break;;
        case 6:
        case 7:
        case 8:
        case 9:
        case 10:
            printf("Between 6 and 10\n");
            break;;
        default :
            printf("Too high (%d)!\n", i);
            break;;
__L0__: break;
  }
```

Figure 3.6: Use of ranges and its translation.

3.2.2 choose statement

The C∀ designers chose to "tinker" with some obscure problems with the **switch** statement, even at the cost of backwards compatibility. Nonetheless, the problems with **switch** are too fundamental, and cannot be resolved without invalidating a significant amount of legacy code. As an alternative course of action, a new control structure, **choose**, is provided as an (almost) drop-in replacement. The **choose** statement takes the form:

⟨*choose statement*⟩ ::= '`choose`' '`(`' ⟨*expression*⟩ '`)`' { ⟨*case label*⟩ }+ ⟨*statement*⟩
  | '`choose`' '`(`' ⟨*expression*⟩ '`)`' '{' [⟨*declaration list*⟩] { ⟨*case clause*⟩ }+ '}'

⟨*case label*⟩ ::= ['`case`' ⟨*case value*⟩ | '`default`'] ':'

⟨*case clause*⟩ ::= { ⟨*case label*⟩ }+ ⟨*statement list*⟩ ['`fallthru`' [';']]

Inside a **choose** block, the start of a new **case** clause signals the exit from the block. The only exception to this rule occurs when the last statement of a **case** is a **fallthru** statement, which provides the same termination semantics as a **case** clause in a **switch** statement.

```
choose (test) {
  case 1:
    /* implicit break, exit block */
  case 2:
    /* explicit fall through */
    fallthru;
  case 3:
    /* ... do something else ... */
}
```

The redundancy of two select statements, although seemingly a departure from the C way of doing things, was introduced in the hope that new programmers would use the new **choose** statement if they are first introduced to it. The more

```
int fred() {
    int i;

    choose ( i ) {
      case 3:
        i = 5;
      case 2, 4:
        i = 3;
        fallthru;
      default:
        i = 3;
    }
}
```

```
int __fred__Fi__()
{
    int __i__i;
    switch (__i__i)
    {
        case 3:
            (__i__i=5);;
            break;;
        case 2:
        case 4:
            (__i__i=3);;
            /* null statement */ ;;
        default :
            (__i__i=3);;
            break;;
__L0__: break;  }

}
```

Figure 3.7: Translation of the choose statement.

natural semantics of the **choose** statement may gradually attract more experienced programmers, rendering the **switch** statement obsolete.

Implementing the **choose** statement is a matter of introducing a simple rewrite in the Abstract Syntax Tree. This rewrite casts every **choose** statement and its corresponding block into an equivalent **switch**, where the implicit **break**s are inserted in their proper places, and the **fallthru** statements are elided (see figure 3.7).

3.3 Exception handling

The ability to exit loops and blocks from various points to a static location naturally generalizes in at least two directions. The first is to extend the ability to jump out of other constructs, in particular, to jump out of function activations. The second direction is a relaxation on the requirement of exiting to a static location. Both features are very useful. Exits from arbitrary code to dynamically determined locations allow the construction of complicated control flow patterns in a structured way. In particular, situations outside the normal purview of an algorithm, e.g., errors, can be addressed consistently without obscuring the real purpose of the code.

In C, abnormal conditions are dealt with (or often not dealt with) by checking special 'error' values from function returns and/or global status flags, or trapping signals, possibly logging the error and terminating execution more or less gracefully. Handling errors by checking the return code of a function has two disadvantages: the error handling code gets intertwined with application code, decreasing the readability of both, and checking for an error can quickly become unmanageable, especially in a complicated function call chain.

Reliable software strives to work correctly, or to determine that correct operation is impossible in a given situation. A mechanism for appropriately dealing with abnormal or exceptional situations must provide:

1. A clean way to cancel the execution of the program fragment within which the situation arose.
2. Means of notifying an abnormal situation has occurred, as well as consistent and expressive ways to describe it.
3. The possibility to match to the situation specific code that knows how to deal with it.
4. The ability to specify what course of action the program must follow once the abnormal situation has been addressed.

Items 1 and 2 are independent of the context: if a code fragment C is unable to guarantee its continuing correct operation, the correctness of any piece of code that depends on C is immediately suspect. In this case, proceeding with the computation is pointless and potentially dangerous, so it is better to cancel the execution of C, and to release any intermediate allocated resources. Also regardless of the context, an exceptional situation has to be described concisely and completely, at the level of abstraction of the code that detects it. By contrast, items 3 and 4 are closely related to the context. For example, it is conceivable that two programs using the same function may want to react differently in the presence of the same abnormal situation. If it is to be useful, a mechanism to deal with abnormalities must be flexible.

An *exception handling mechanism* (EHM hereafter) constitutes a family of programming language constructs that furnish the programmer with linguistic means to address all the concerns outlined above. Despite reservations by a number of prominent language designers (among whom are names of the caliber of C.A.R. Hoare [67] and Doug McIlroy[1]), most modern programming languages include an EHM. EHMs allow a clear separation of the "usual" and exceptional control flows, and do away with the need of multiple testing for the same error and *ad hoc* control transfers.

Although EHMs have been proposed for a variety of runtime systems, this discussion assumes a procedural, non-object-oriented, sequential language in which functions' activations are organized in a single stack (i.e., a C-like environment). This stack is assumed to grow upwards, i.e., when a function invokes another, the activation of the second function is assumed to be on top of the first. This convention is also observed in the figures that illustrate this chapter.

Different EHMs have been extensively studied, but there is no universally-accepted terminology. This work uses terminology from Buhr et al. [22]. An "exception" is characterized as an event that is ancillary to algorithmic execution, and which is comparatively infrequent. When a code fragment detects the occurrence of such an event, it can notify the rest of the system by *raising* or *throwing* an *exception*, which is a data object describing the event. An EHM permits exceptions to be structured via *parameterization* and *derivation*, providing expressive means to describe exceptions across orthogonal concerns. To address issue 3 above, EHMs introduce the concept of a *guarded region* with which code specific to the kind

[1]McIlroy claims that exceptions cause a system to be *less* reliable, as programmers and library writers throw exceptions rather than try to understand the problem, or even report it in a concise and complete manner [122].

of exception is associated, and organized in *handlers*. Blocks without associated handlers are called *unguarded regions*. What handler is activated (or *catches* the exception) in each particular raise is determined by the *propagation model*. When one of the intervening handlers completes successfully, the exception is said to have been *handled*, and the system proceeds according to the EHM's *transfer model*, which could be *termination* or *resumption*.

In the following section, the design space of the components of an EHM is explored in some detail. With this background, C∀'s EHM design and implementation is described in the next section. Finally, alternative decisions to incorporating exceptions into C are briefly reviewed as related work.

3.3.1 EHM design space

Canceling incomplete operations

During propagation, multiple stack frames (activation records) may be terminated. Each stack frame contains a function's arguments and local variables among other data. All local data must be cleanly deleted. Resources, like dynamic memory, file descriptors, socket descriptors, etc., may have been acquired during initialization of locals or in the course of the completed part of a function, and it is necessary to return them to the pool of available resources. EHMs usually guarantee the correct termination of automatic data, and provide, either by themselves or in conjunction with other language features, a means to free manually allocated resources. Examples of EHM constructs that perform this service are **finally** clauses in Java, or unwind−protect special forms in Scheme [116] and Common Lisp. In C++, this effect can be obtained by using a destructor for a class.

Describing exceptions

The correct description of an abnormal situation is crucial to the determination of the action to be taken in response. EHMs in languages like PL/I, Ada or Lisp describe abnormal situations by tags, strings or a system-wide numeric encoding.

This approach is clearly not extensible, or even convenient in larger systems. An alternative approach, taken by ML, is to describe exceptions by *exception types.* ML's algebraic types (disjoint-tagged-unions) serve this purpose admirably. Furthermore, having types describe exceptions allows the programmer to pass additional structured data along with the name of the exception, giving raise to *parameterized exceptions.* Since exception types and "computational" types are used for quite different purposes, they are usually kept separate from each other [25]. Languages like C++, however, allow the programmer to throw objects of *any* type, which confuses this difference.

C++ and other object-oriented programming languages further refine the "exceptions are types" approach by organizing exception types in hierarchies and allowing the propagation process (described below) to match exceptions with handlers for ancestor types. This feature is usually known as *derived exceptions*, and is probably one of the clearest uses of inheritance as a classification device in the classical sense [104]. It is also one argument in support of multiple inheritance. It is common that exception types form the deepest hierarchies in class libraries for languages like C++, Java or C#.

The exceptions a function potentially raises are an aspect of its behaviour. Function behaviour is usually encoded in an interface specification (also known as *prototype* in C/C++ or *signature* in Java), so it is natural to extend such specification to include exceptions, resulting in another feature common in EHMs: *exception lists.* These lists are most noticeable in Java. The exceptions a Java method can raise are enumerated after its parameter list[1]. Any code invoking a method so qualified must either catch all the exceptions in the method's exception list, or include the ones that go uncaught in its own exception list. This allows the Java compiler to check for the compatibility of methods, and serves as documentation on the function's operation. In practice, however, most programmers feel that exception lists violate the principle of information hiding, cause more problems than they solve (Should an overridden method throw the same exceptions as the method it overrides? If

[1]This is only true for *checked exceptions* (those that inherit from Exception). Other kind of exceptions, namely run-time exceptions (inheriting from RunTimeException) do not decorate the signature in this way.

the thrown exceptions are to vary, need they do so covariantly?), or overwhelm programmers with details. This state of affairs often results in exception lists being overly general, and exception lists only containing the supertype java.lang.Exception are common. This practice discards whatever error-specific data the exception carries and in so doing defeats the purpose of parameterized exceptions. Exception lists are also problematic in generic code, as the requirements on the functions a type provides must also list the exceptions those functions potentially raise. As stated above, exceptions are a concern *orthogonal* to the notion a method, generic or otherwise, implements. Having the exception specification play such a prominent rôle, for all its usefulness, is, in commonly used type systems, highly inconvenient; there seems to be no middle ground.

Guarded regions

Handlers that deal with specific exceptions are associated with guarded regions. The granularity of a guarded region can range from a subexpression [60] to blocks of statements, and without loss of generality, most statement-oriented languages use a compound statement. This design allows lexical nesting of guarded regions. As the program runs, and control flow proceeds through guarded regions, function invocations also cause guarded regions to dynamically nest. Upon entering a statically or dynamically nested guarded region, the guarded region's handlers are added to the list of available handlers. Similarly, when control crosses the borders of a guarded region on its way out, the handlers associated with the region are removed from the list of available handlers.

Propagation

As noted by item 4 on page 44, an EHM determines the control flow the program takes when an exception is raised. This control flow generally involves accessing one or more handlers, and, after the exception is handled, transfers control to a point where normal operation of the program can continue, or the program can safely terminate.

Complex control flow patterns, especially across binding environments, are usually explained by describing control transfers in terms of functions. Jumps, for example, are presented as functions that never return, and exceptions are described in terms of *one-shot downward or upward continuations* [130]. A dual approach, taken in [22], decomposes a function, inasmuch as control transfers are concerned, into two jumps, a *call* and a *return*. If it is possible, by using the lexical structure of the program, to determine statically, i.e., before the program is run, the symbolic address control transfers to by a jump (call or return), the jump is qualified as *static*. In contrast, if a particular control path of the program is the basis for the determination of jump targets, the jump is *dynamic*. The possible combinations of static/dynamic call and return generate a taxonomy that is useful to characterize exception propagation models. All four cases are discussed.

		Call	
		Static	Dynamic
Return	Dynamic	1 function call	2 resumption
	Static	3 sequel	4 termination

Table 3.1: Control Structure Taxonomy

1. Normal function invocation in a lexically-bound environment transfers control to the address of a lexically-visible function (static call). The code for the function is executed, and, upon completion, control returns to the point immediately after the point of invocation, which cannot be known until the code is run (dynamic return).

2. A function call in a dynamically-bound language, e.g., Emacs Lisp [96] transfers control to the function definition that is closest in the call chain to the point of invocation, a location not determinable until runtime (dynamic call). Similarly, dynamic propagation models also rely on dynamic call (raise). Handler selection depends on the nested list of available handlers built at runtime, allowing handlers to vary for each function call. The code of the selected handler is executed, and, upon completion, control returns after the raise point (dynamic return). This behaviour constitutes the *resumption transfer model*.

3. Tennent [129] proposed a construct, called a *sequel*, using static call, but that returns to the point past the end of the lexical scope where it is defined (static return). Knudsen [80] built Beta's EHM around this construct, but this design seems to have few proponents.

4. The most popular combination for EHM is dynamic call and static return because of its flexibility. Handler selection is done as described in item 2. Once the selected handler has completed, control continues in the lexical context after the guarded region associated with the selected handler. This behaviour constitutes the *termination transfer model*.

Dynamic call is almost universally considered the right design decision for an EHM. However, the right choice between dynamic or static return (termination or resumption) is less clear. Since they are not mutually exclusive, it is possible to adopt both termination and resumption models in a programming language. Both models are depicted in figure 3.8. The call stack is shown on the left, growing upwards, where every block denotes an activation. If there are handlers associated with an activation, they are shown on the right. The solid arrows connecting activations denote (normal) flow of control. Upon the raise of an abnormal event (marked "raise"), the propagation mechanism searches among the available handlers for one that is appropriate to the raised exception. This search is denoted in the figure by the dashed line through the handlers on the right. In the situation depicted in the figure, there is no appropriate handler to the raised exception among those associated with the tightest (lexically- or dynamically-) enclosing guarded region, so the propagation continues among the handlers at the next guarded region. A matching handler is found among these (marked "catch"), so the handler code is executed. At some point in this process, the handler's code determines that the abnormality is better addressed at a higher level, so it *reraises* the exception (marked "reraise"). This causes another round of propagation of the same exception to begin, this time among the handlers associated one level out (notice that the dashed line does not pass through the last available handler on the current level), and not stop until another appropriate handler is found two levels out (marked "catch"). This handler's code is run to completion. At this time the exception is considered *handled*.

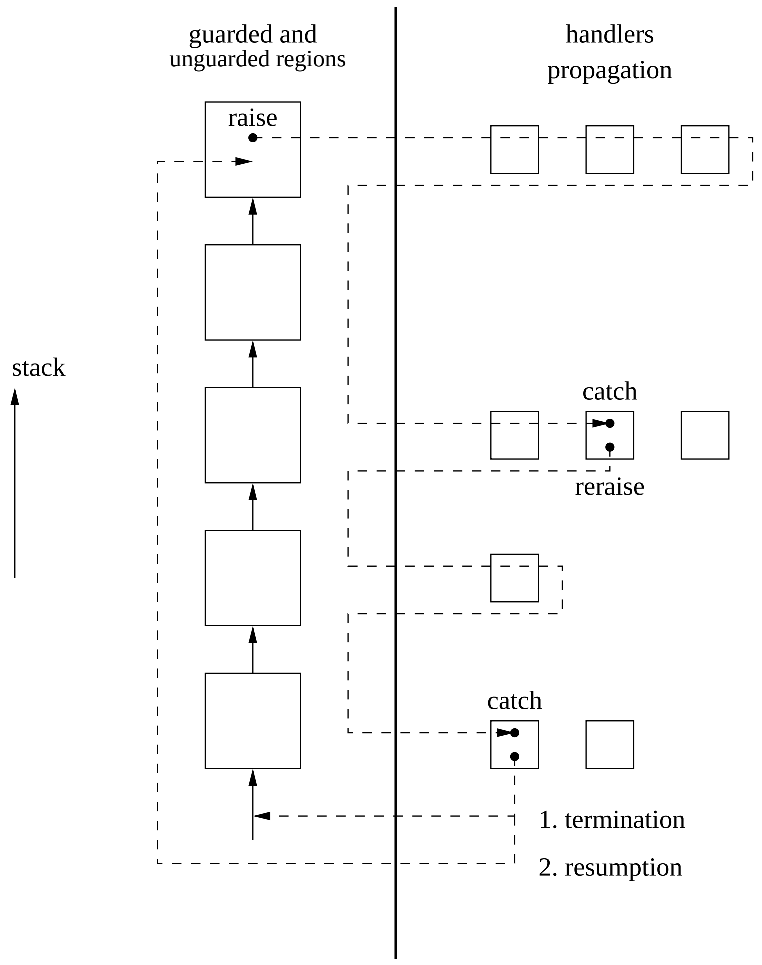

Figure 3.8: Exception handling

The handler-clause matching usually proceeds in accordance to the rules already in place with respect to other aspects of the language, e.g., type equivalence, or the extended rules of argument-parameter matching. These rules govern also the possible ways the exception can be used in the body of the handler, e.g., what operations can be applied to the exception object. A common extension to the function invocation analogy in handlers allows for the provision of a default or "catch-all" handler, that matches *any* exception.

What happens after the exception is handled is determined by the policy in whether the return location is static or dynamic. The mechanism can choose a *sequel*-like behaviour (as in Beta) and transfer to the end of the guarded region that handled the exception (point "1" in figure 3.8), resulting in *termination* semantics. Alternatively, if dynamic return behaviour is chosen, control is returns to the point after the exception was raised (point "2" in figure 3.8), effectively resuming the operation within which the exception occurred, resulting in *resumption* semantics.

Termination-versus-resumption is an ongoing debate in programming language circles, with resumption most favored among the Xerox PARC-bred community (Smalltalk, CLOS, Mesa/Cedar), and termination advocated among the C++ and Java camps. One model does not preclude the other, and it has been suggested that they are complementary, and serve different purposes, and address different concerns. If the two models are to coexist in a language, however, close attention has to be paid to several issues. Foremost among these is *when* is the propagation model for a particular raise selected. There are three possibilities:

1. As part of the exception type (in the declaration).

2. Indicated at the raise point.

3. Indicated at the handler.

Encoding propagation model preferences in the exception type should be flexible enough to provide for the case when the exception is *dual*, i.e., when it is not specific to either mechanism.

Regardless of what propagation model is chosen, the combination of resumable and terminating exceptions calls for more sophisticated handler semantics, that regulate the coupling or the propagation model specified by the raised exception and the handler that is to deal with it. This consideration is part of the *handler matching* semantics, and is usually specified by describing the behaviour of the system under all possible combinations of exception/handler choice of propagation models. Not all such combinations make sense. In particular, it is impossible to match a terminating exception with a resumption handler, since by the time execution is to be resumed, the stack has been unwound. In general, unmatched exception/handler combinations are problematic, and the most conservative semantics for these cases are usually safest.

The added flexibility of dynamic propagation is not without drawbacks. In particular, the case of an exception going uncaught and escaping to the runtime, due to a lack of an appropriate handler. Different programming languages handle this case differently. Some, like Java, try to determine statically whether there is the possibility of this happening in a particular program, via powerful control-flow analysis at compile time. C++ provides a **terminate()** function hook for uncaught exceptions, whose default behaviour is to abort the execution of the program. Despite this important failing, dynamic propagation remains the most popular choice.

3.3.2 C∀ Exception Handling Model

In this section, the syntax and semantics for the C∀'s EHM are presented, relying on the background developed above. The C∀ EHM includes parameterized and derived exceptions, as well as both resuming and terminating transfer models.

Describing exceptions

In contrast with C++, where values of any type can be used as exceptions, in C∀, the only type that can be thrown is `exception`, an alias for the type **struct** `_exception`, defined in the runtime library. Derived exceptions are constructed from this type via record composition. That is, more specialized exception types are `structs` that include an anonymous field of type exception. This rule builds upon the implicit conversion rules in C∀'s static type system (inspired by a similar extension in the Plan-9 C compiler [99]), which introduces an implicit conversion from **struct** instances to the type of an unnamed field:

```
typedef struct _exception { char *msg; } exception; /* runtime library */

struct _io_exception {
  exception;
  char *device_name;
} io_exception;

printf("Exception: \%s", io_exception.msg); // implicit cast to _exception
```

This approach, taken recursively, permits forming exception hierarchies *à la* object-oriented inheritance chain, rooted on exception. In this way, more organized exception handling patterns and some reuse of handler code are possible.

Guarded regions

Like C++ and Java, guarded regions are compound statements prefixed by the keyword **try**. Inspired by Java, C∀ includes a **finally** clause that is executed regardless of how the **try** block terminates. Every guarded region may have multiple handlers associated with it; each handler comprising a compound statement and introduced by one of the keywords **resume** or **terminate**. There is at most one **finally** clause after the handlers:

$\langle \textit{try statement} \rangle$ ::= ‘`try`’ ‘{’ $\langle \textit{statements} \rangle$ ‘}’ [$\langle \textit{handlers} \rangle$+] [$\langle \textit{finally clause} \rangle$]

$\langle \textit{handlers} \rangle$ ::= [‘`terminate`’|‘`resume`’] ‘`(`’ $\langle \textit{exception spec} \rangle$ ’)’ ‘{’ $\langle \textit{catch clause} \rangle$ ‘}’

$\langle \textit{finally clause} \rangle$ ::= ‘`finally`’ ‘{’ $\langle \textit{statements} \rangle$ ‘}’

Notice that C∀ follows the C++ approach for the scope of handlers and **try** blocks. By requiring handlers to be associated with blocks, the language syntax effectively prevents declarations in **try** blocks from leaking into handlers. In particular, it prevents the problem of the handler referring to objects that are nonexistent. Consider, for example, the case of a declaration placed after an expression that raises an exception.

The *exception specification* is a C∀ variable declaration with certain restrictions. In particular, since C∀ uses name equivalence for types, declaring a new type in a handler, e.g.:

```
try {
...
} terminate( struct _new_exception { exception; /* more fields */ } e) {
...
}
```

introduces code that is effectively unreachable. The current implementation of the C∀ translator flags this situation as invalid. It also makes sure that the catch-all **terminate**(exception) or its **resume** counterpart occurs at the end of a handler list.

Raising and propagating exceptions

An exception can be raised at arbitrary points in the program. The syntax for raising (and re-raising) an exception is:

$\langle$*raise statement*$\rangle$ ::= ['`terminate`' | '`resume`'] [$\langle$*expression*$\rangle$] '`;`'

As can be seen, the propagation model for a given exception is set both at the raise and handle points. Encoding the propagation model in the exception type involves modifications to the type system (**terminate** can only throw terminating exceptions, etc.) and imposes finer control over the "inheritance-as-composition" mechanism, for no expressive gain.

C∀ has *matching semantics* of handlers, that is, a handler has to match the exception type, specified in the handler clause, *and* the propagation model, specified by the kind of raise/handler pair, for it to be selected. A reraise must match with the propagation kind of the initial raise.

When there are no raised exceptions, control flows through guarded regions, ignoring handlers and executing **finally** clauses. When an exception is raised, the propagation mechanism searches the guarded region for available handlers lexically from top-to-bottom for each handler clause, in the order of definition. The first handler that matches the raised exception is the one selected. Note that this strategy means that a more general exception specification can shadow a more specific one occurring later in the handler list, thereby rendering the latter handler effectively unreachable.

Upon handler selection, the exception parameter (the result for the expression of the raise statement) is bound to the variable declared in the exception specification parameter.

Assume that the selected handler runs to completion, i.e., that no further exceptions are raised or reraised during its execution. For a **resume** handler catching a resumable exception, the execution state at the point of the raise is restored, and execution continues at the statement after the one that issued the raise. For a **terminate** handler catching a terminating exception, the execution state between the raise point and guarded region containing the selected handler is discarded, and control continues after the guarded region, once the **finally** clauses of all discarded guarded blocks are executed.

If after the stack of available handlers is completely examined an appropriate handler is not found, the uncaught_exception function hook is invoked. The default behaviour of uncaught_exception is to abort the execution of the program, but a programmer can provide an alternative implementation. However, an alternative implementation must eventually abort.

3.3.3 Implementation

There are two popular strategies for the implementation of a termination-based exception handling mechanism. Christiansen [28] calls them *dynamic registration* and *static table*. The static table approach is only feasible when access to the (machine) code-generation engine is possible, as knowledge of the addresses (symbolic or otherwise) of generated code is necessary. For a source-to-source translator, this approach is clearly infeasible.

For my implementation of C∀'s EHM, I selected the dynamic registration method. This approach consists of storing the state of the program[1] upon entry to a guarded region in a LIFO list, or stack. The state information is stored in a record that also contains a flag describing the status of the propagation. A runtime library function creates and links records onto a *guarded-region stack*. This function is invoked upon entering a guarded region. Also, a guarded block is translated into a **switch** statement whose cases reflect the current status of the propagation, i.e., whether a handler is being searched, or the **finally** clause is being executed. The handler clauses are placed in one of the cases and the **finally** clause in the last. If the guarded region's run is completed normally, the record is unlinked, and the **finally** clause, if present, executed.

When an exception is raised, is top of the guarded-region stack is examined for the next available group of handlers, i.e., the ones corresponding to the closest dynamically enclosing guarded region. As pointed out above, C∀ matches handlers to exceptions depending on the exception type and then confirms the match by ensuring that the propagation model for the raise and handler specifications are the same. Runtime type matching is performed using a runtime type descriptor, or RTTD[2]. For the C∀ translator, the RTTD is a linked list of names, containing the name of its declared exception type and a list of the exception types it is derived from (cfr §3.3.2) recursively. This representation permits the type matching to take the form of a linear search over lists. The translator converts handler clauses into **if** chains, preserving the order of handlers and guaranteeing that the first handler that matches is the one that is selected.

If a handler is not found, control is transferred to the **finally** code, if it exists.

[1] *Ad minimum*: the values of the stack and frame pointers and the program counter.

[2] Common RTTDs are strings containing type names, perhaps after mangling. Full fledged, dynamic type identification (RTTI) schemas, like that of the GNOME GObject framework [128] are also common.

If this code is run to completion, the top record on the guarded-region stack is unlinked, and the process proceeds with the next record. If a handler is found, its code runs. If this code is run to completion, control is transferred to the **finally** clause, with the subsequent unlinking of the top record once the **finally** code is done. The exception then is handled, and the program proceeds at the level of the catching guarded region.

The standard C library functions for nonlocal jumps, `setjmp` and `longjmp` provide for jumps downward in the call stack, and automatically unwind the stack. Thus, this facility does not allow for the implementation of resumable exceptions. More versatile, but less portable libraries that provide the necessary functionality are POSIX [69] user-level context-switching functions[1]. In particular, operating systems like Windows and even several Unix variants like FreeBSD are lacking in these facilities. Even with compatibility among operating systems, the problem is not completely solved (for example, at the time of this writing Cygwin still did not include the `ucontext` functions). Using `ucontext` functions permits the storage of the program state at the raise point of a resumable exception, state that is restored once the exception is handled. As a result of these technical difficulties, resumption is not implemented in the current C∀ translator.

The translated output for a program containing terminating exception handling code is presented in detail in figure 3.9.

The implementation described above is not without problems, the most obvious of which is performance. Each **try** block has to set up a new node in the guarded-region stack. If no exceptions are thrown, but there is a **finally** clause present, `longjmp` still has to be called, incurring a performance hit. Furthermore, the GNU implementation of the non-local jump facilities make extensive use of the C++ runtime system, in particular the exception-handling support routines. Most of the code in these support routines has to do with ensuring that all objects with a local scope are destroyed (and their destructors invoked) when the corresponding scope is abandoned, which is not needed for the purposes of C∀ exceptions. Us-

[1]Other possible choices include the `libunwind`[5] library, that defines a portable API for call-chain manipulation in C programs. At the time of this writing, it supports only Itanium and x86 architectures, which is not portable enough.

```
int foo() {
  io_exception nofile;
  stcpy(nofile.msg, "No file found.");
  terminate nofile;
}
//...
void bar() {
  //...
  try{
    //...
    foo();
  } terminate( io_exception ex ) {
    // handler 1
  } terminate( exception ) {
    // terminate any
  } finally {
    close(fd);
  }
}
```

```
int foo() {
   {
     excobj.data = (void *)&x; // global object
     longjmp();
   }
   /* alternative execution path */ return 0;
}
//...
void bar() {
 {
  gd_region_link link;
  gd_add_context( &link );
  switch( setjmp(link.ctx) ) {
  case 0: {
   //protected region of code, including:
   foo();
   longjmp( link.ctx, INHANDLER );
  }
  case INCODE:
   // beginning of handlers
   if(match( excobj.data_type, handler1.type )
      & excobj.prop_model == handler1.prop_model )
      // handler 1
      io_exception exc = (io_exception)excobject.data;
      // ...
      excobj.handled = true;
      longjmp( link.ctx, INHANDLER );
   } else
     if(excobj.prop_model == handler3.prop_model)
     {
       // catch-all handler
       excobj.handled = true;
       longjmp( link.ctx, INHANDLER );
     }
  case INHANDLER:
  {
   close(fd);
   gd_remove_context(&link);
   if ( ! excobj.handled )
    longjmp( gd_top_context(), INCODE );
  }
 }
```

Figure 3.9: Exception handling translation.

ing compiler or library-specific facilities, such as GNU C's __builtin_setjmp() and __builtin_longjmp() that limit their functionality to basic computer state information might significantly alleviate the performance degradation.

Furthermore, there are concerns about the generated code. Global variables are used to store the root of the guarded-region stack and the exception data object, which is almost always poor practice, and immediately problematic when concurrency is introduced to the program.

3.4 Related work

The addition of control structures to the C language has been the subject of many an academic and technical study. Most of them involve modifications to the runtime organization (for example, Budd added Icon-style generators [21], a work which involved substantial reorganization of the runtime stack).

Modifications to the **switch** statement and the default fall-through behaviour have also been occasion of much discussion. Of these studies, it is worth singling out Cyclone [127], which extends **switch** to handle values of any type and **case** labels to specify *patterns*, possibly qualified by *guards*, which are predicates whose truth is required for the pattern to match, and the particular **case** to be selected. Particularly interesting is the fact that **switch** conditions and **case** patterns in Cyclone may involve *tuples*. Cyclone also forbids falling through non-empty **case**s, and introduces the keyword **fallthru** for the programmer to confirm that fall through is the desired behaviour.

EHMs have been integrated into C in a great variety of forms. The dynamic registration method has been especially popular in this task for it implies little or no changes to the compiler or runtime. The most complete description of the dynamic registration method is by Cameron et al. [27], in the context of C++. Examples of this approach are the `cexcept`[34] library, that provides C++-like **try**, **catch**, and **throw** constructs; and the real-time oriented RTFfiles [111], that also includes **finally** clauses. Neither of these libraries permit resumption, or even matching by type, as exceptions are identified by system-wide numeric codes. Another exponent of the dynamic registration approach, also worthy of mention is Allman's [8], which uses string exceptions in combination with regular expression matching at the handlers for a flexible form of derived exceptions that does not involve inheritance or inheritance-like mechanisms.

For better performance, platform-specific C extensions, like those known as "Structured Exception Handling"[98] on Windows platforms could be generated. At a lower level, if C is not to be generated, but an intermediate code is targeted, Ramsey and Peyton-Jones's C– [102] provides versatile exception handling mechanisms, and is coupled with a variety of backend code generators. At an even lower level, the most popular virtual machines to date, Sun's JVM and Microsoft's CIL facilitate the walking of the stack, which allows for very flexible EHM implementations. At the architecture level, very few platforms (e.g., SPARC and MIPS) support the new generation of generic stack unwinders.

Alternative approaches to error handling for C, besides the ones mentioned at the start of the section, are basically extensions to the status-flag technique. An example of this is an approach used by most C-CORBA bindings [63], consists of passing extra by-reference "environment" records as arguments to functions. The programmer needs to check the values the function has placed in specific fields of this records to determine whether an error occurred during the functions' run. Although this is perhaps the most appropriate way to check for errors in a distributed context, it suffers from the same weaknesses as the status-flag methods.

Chapter 4

Tuples

The concept of "subprogram" arose in the early days of computer programming. In general, a subprogram is a named, self-contained, possibly parameterized, fragment of code that performs some well-defined action that is invokable from another subprogram, to which it may or may not return a value. The key point is that a subprogram is independent of its caller, therefore usable by any other subprogram at any point in its code.

The subprogram is the most common abstraction mechanism in all programming languages, and was present even in the first one, Fortran. Subprograms play a pivotal rôle in the "structured programming" school of thought, and in software engineering with respect to the notion of modularization. Today, virtually every programming language supports the notion of a subprogram and all programmers are taught about and use subprograms as a primary coding mechanism.

Under the concept of subprograms, endless variations of the original theme can be found: arguments can be passed to the called subprogram according to a number of policies (as copies, references or a representation of their actual textual form), argument lists can be of any fixed or even of unspecified length, subprograms can directly or indirectly call any other subprogram including themselves *ad infinitum*, can be called asynchronously, or called on a different computer, etc.

It is perhaps surprising that these numerous mutations, generated over the more than 40 years of research and use, have not wandered far in form and concept from the original notion of subprogram call. The main components of such a call are still clearly recognizable: a list of arguments passed to parameters, a change in the control flow from the call to the subprogram, saving the necessary execution state so it can be restored after the subprogram and those that it has called have finished; and, should it be necessary, the placing of the result of the subprogram computation in a location where the calling program can access it.

Exploring the design space depicted above, KW-C [131], one of the direct forerunners of C∀, included output parameters, named return values, and other features. Most notable among the extensions, it introduced to C language the idea of *tuples*. Tuples capture in a limited way the notion of independent computations, and thus are more a data structuring mechanism than a new built-in type; tuples allow for a more concise expression of several idioms in C that involve the use of temporaries. These idioms include multi-valued functions, the manipulation (packing and unpacking) of composites, operations like the simultaneous selection of multiple members from records, or initialization of the same, and various parallel forms of assignment. Having tuples in the language establishes uniformity to all such manipulations.

C∀ relies on tuples to enhance the C∀ language described by Ditchfield [47] and Bilson [16] in much the same way that KW-C does for C. To this end, I extended the original C∀ expression analysis algorithm to include the effects of overloading and type specialization in tuple operations. Moreover, I generalized KW-C's notion of a tuple to encompass *designators*, a feature that allows for very expressive function composition patterns. Other features that resulted from the extended expression analysis phase are named parameters, default values for arguments, and named return values. The description of these modifications, the features they sustain and, more importantly, the resulting increased expressiveness of the language form the substance of this chapter.

4.1 Multi-valued Functions

In the most general case, a subprogram accepts arbitrary number of arguments of arbitrary types and returns arbitrary number of values of arbitrary types. There are a number of possible explanations for programming languages designers not giving multiple value-returning (MVR) functions the same importance (if any at all) as the single-valued returning (SVR) counterpart. First there is the notation. Awkward syntax is probably the first reason that discourages designers from including MVR functions in their language. Ideally, a MVR function should be syntactically as similar as possible to a SVR function, that is, it should be possible to declare and use an MVR in every context where a traditional SVR function occurs, specifically:

returning values: It should be possible to store the values returning from a function into appropriate variables, and

composition: It should be possible to use the values returned from a function as arguments to another (provided they are type-compatible), without the need of intervening temporaries.

Syntax meeting these requirements is not immediately obvious in most programming languages. Particularly problematic is the *composition* of MVRs. For example, given a (curried) function type ascription, expressed as:

$$\begin{aligned} &\text{fun} f : \text{Type}_{f,1} \rightarrow \text{Type}_{f,2} \rightarrow \ldots \rightarrow \text{Type}_{f,n} \\ &\text{fun} g : \text{Type}_{g,1} \rightarrow \text{Type}_{g,2} \rightarrow \ldots \rightarrow \text{Type}_{g,m} \end{aligned}$$

if $\text{Type}_{g,i} \ldots \text{Type}_{g,k}, i, k \leq m, m \geq n$, match $\text{Type}_{f,n-k}, \ldots \text{Type}_{f,n}$, the functions f and g are composable, and their composition has the type ascription:

$$f \cdot g : \text{Type}_{f,1} \rightarrow \text{Type}_{f,2} \rightarrow \ldots \text{Type}_{g,m}$$

If SVR-function composition is expressed in prefix notation $f(g(\ldots))$, it is clear from the denotation that the result of the innermost function is to be used as the argument of the next function in the chain. For MVR-functions, however, it is not so obvious what results are to be bound with what parameters. An infix 'apply' operator solves the problem:

$$(\text{Value}_{Type_1}, \text{Value}_{Type_2}, \ldots \text{Value}_{Type_n})\mathbf{app} f \mathbf{app} g \tag{4.1}$$

This form of notation is so appealing in this particular case that even otherwise algebraically-inspired languages, such as Beta [87] adopt it (the proposed **app** operator is written `=>` in Beta). The function `takes_two` below takes two parameters, called *input parameters*[1], which are assigned (left-to-right) from the variables `y` and `z` as arguments:

```
y,z => takes_two
```

The MVR function `gives_two` returns two results. Function composition is then performed as follows:

```
gives_two => takes_two
```

This syntax allows the function composition in formula 4.1 (for $n = 2$) to be expressed as:

```
val1,val2 => f => g
```

Beta's choice of notation certainly looks familiar to programmers of the Unix shell scripting languages, which rely heavily on program composition, and where communication takes place exclusively with text streams. Connecting the output of a program to the input of another (the **app** operator above) is done using a 'pipe' character (|). Raoult and Sethi [103] propose incorporating the pipe notation into a compiled programming language.

A second syntactical alternative, postfix notation, makes composition of multiple-value returning functions unfettered in stack-based languages, where the symmetry between multiple arguments and multiple return values has been commonplace. Consider the Forth version of the running example:

[1]also part of KW-C.

```
: \gives_two 2 3 ;
: \takes_two {a b} ... ;
\gives_two \takes_two .
```

The first two lines define the functions ("words" in Forth terminology) `\gives_two` and `\takes_two`. The first one pushes the values 1 and 2 onto the stack. The function `\takes_two` pops two values from the stack and places them in the variables `a` and `b`. The last line is an expression composing both functions.

For languages using a prefix function application operator things become more difficult. Most languages in this category have been unsuccessful in achieving nearly as natural a form as those outlined above and have to resort to alternative syntax (as Beta did). Consider, for example, Scheme, which, as of the last revision of its standard [76], includes MVR functions. However, functions returning multiple values have to follow a particular interface, namely, generating a group of variables with the special form `values`, and calling a function upon an expression generating multiple values with the form `call−with−values`, which takes a closure with no parameters as a *producer*, and a second closure, the *consumer*, which is called with the values generated by the producer as arguments. Although certain optimizations that elide the use and overhead of these closures are possible [12], calls to MVR functions do not resemble applications of a SVR function. They look awkward in even simple cases, such as when using a recursive definition:

```
(define partition
; 'partition' takes a list and a predicate, and returns the list of all the
;     elements of the original that comply with the predicate, and the list
;     of all the elements which do not.
  (lambda (l p)
    (if (null? l)
        (lambda () (values '() '()))
        (call−with−values (partition (cdr l) p)
          (lambda (lyes lno)
            (if (p (car l))
                (lambda () (values (cons (car l) lyes) lno ))
                (lambda () (values lyes (cons (car l) lno) ) )))))))
```

John McCarthy was among the first language designers to recognize the importance of MVR functions, and urged the american delegation in the Algol design committee to include this facility in Algol 60, and made it part of even the first incarnations of Lisp. Later, Friedman and Wise [55] extended Lisp to incorporate easier-to-use recursive MVR functions (a facility that is almost as expressive as C∀'s). It is not surprising then, that later incarnations of Lisp, in particular ANSI Common Lisp ([61]), with its emphasis on pure functional programming style, frown upon the use of 'out' parameters and provides a special facility for the manipulation of MVR functions. Any function can return multiple results that are bound via special forms and macros: `values` returns its arguments (without intervening structuring), `multiple−value−bind` names these values on the receiving end. This approach increases readability, but function composition is still awkward:

```
(labels ( (gives−two () (values 2 3)) )
  (multiple−value−bind (x y) (gives−two) (takes−two x y)))
```

The form `multiple−value−bind` binds the two values returned by `gives−two` via `values` to the names `x` and `y`. `multiple−value−bind` is but one of the specific forms of `destructuring−bind`, which matches more structured (nested) lists. Once the variables are bound, they can be used as arguments to call `takes−two`.

Other Lisp-inspired languages, even with more algebraic syntax also make use of a similar interface for dealing with multiple values from a function. For example, Dylan [113], a hybrid between Scheme and CLOS comes closer to the objectives stated above. It also makes use of the **values** special form, but reuses the local assignment form **let** to make use of the values returning from a function.

```
  define method gives−two ()
   => ( a :: <number>, b :: <number> );
    values( 2, 3 );
  end method
⋮
  let ( x, y ) = gives−two();
  takes−two(x,y);
```

For Algol derivatives other forms have been proposed. Languages like Xerox's Mesa [91] and its successor, Cedar, treat functions as if both parameter and return values lists were record structures. When assigning the result (or rather, the container for the result), any variable whose structure matches the one returned by the function is type-compatible.

```
gives_two: procedure returns [a,b:integer] =
begin
   return [a:2, b:3];
end;

⋮
[ x, y ] = gives_two[] ;
takes_two[x,y];
```

This interpretation of parameters and return value lists as records is used to some extent in C∀(see page 81).

Other languages that provide for multiple-value returns are C's forerunner, BCPL [105], Alphard [114] and CLU [88]. While all the algebraic languages above deal with returning multiple values, none addresses the issue of function composition of MVR functions consistently with their SVR counterparts.

Since all these incarnations of the mechanism are unappealing or do not integrate well with the rest of the language (e.g., inconsistencies between single- and multiple-value returning functions), alternate solutions to returning more than one value from a function are usually preferred. The effect of returning multiple values from a function can be simulated to some extent by a combination of other programming language constructs (aggregate or 'out' parameters [1]) or by resorting to programming conventions, like rewriting a program in continuation-passing style. These workarounds carry their own set of difficulties.

[1]'Out' parameters can be used to trigger optimizations, for example, by returning multiple results in registers (as it happens in languages like Ada, Sather and Mercury). If unavailable in a particular language, the effect of 'out' parameters can be approximated by passing extra arguments by reference.

Passing aggregates back and forth has the inconvenience of requiring cumbersome and error-prone packing and unpacking. Output arguments and passing parameters by reference require additional notions such as variables, addresses, assignments, nonlocal side effects, aliasing, etc, that place a considerable burden on the programmer.

Furthermore, any of these approaches involves the use of temporary variables whose sole purpose is to immediately transfer the results from one function to another. This use of variables is undesirable from a number of standpoints. Firstly, it increases the *data complexity* of the function call that makes use of such a practice. Data complexity is a measure of the amount of data processed by a subprogram, and it is reflected, among other criteria, in the the number of variables declared therein [32]. Secondly, it multiplies the places that require change should any of the functions involved change its interface[1]. Lastly, for languages like (pre-C99) C, all variables must be declared at the beginning of a block, which can be substantially separated from the point where the function call is made (unless "spurious" blocks are introduced by the programmer, an unsatisfactory solution), inhibiting readability and maintainability. Languages like C99, C++ and Java alleviate this situation by allowing the declaration of variables to be interspersed with the statements but the variable count is still artificially (and awkwardly) increased.

Continuation-passing style (CPS), and later the *CPS-transform* was first introduced by Fischer and Plotkin [53, 100] and extended by Harper et al. [65] to the typed case. It does not involve the use of temporaries, but entails its own set of difficulties. In its original form, CPS makes use of a reified form of the program state, called a *continuation*. A continuation is a function that represents the "rest of the program", i.e., the computation that is to take place once the current computation is done, and to which the result of the current computation potentially contributes. Roughly described, CPS involves passing a callback, representing the continuation, to every function. At the end of its computation, and instead of returning a value

[1]In the best of cases the temporary and the new interface would be incompatible, so an error is detected by the compiler. However, it is more likely that an implicit conversion and potential information loss would take place, leading to hard-to-find bugs.

to its caller, the called function invokes the callback on the values it is to return. For the running example, the use of these technique looks like:

```
void takes_two(int a, int b, ... );
⋮
void CPS_gives_two( void (*callback)() ) {
  int ret1;
  double ret2;
⋮
  (*callback)(ret1, ret2);
}
⋮
CPS_gives_two( takes_two );
```

This method transforms the problem of returning multiple values to the accepting of multiple arguments, and thus avoids the creation of temporaries. However it requires a complete restructuring of the code, not always possible in a separate-compilation setting.

In any case, all the approaches described in the preceding paragraphs hide the fact that what the programmer wants to express is the composition of functions.

4.1.1 Importance of MVR functions

MVR functions make the code amenable to various optimizing transformations. In particular, since the returned values are independent of each other, and are passed unpacked, they could be passed back in registers rather than using the stack, providing some speedup. Also, using more information on how the returned values are going to be used (as arguments to a function, or assigned to new variables) would conceivably allow for closer, more efficient caller/callee interaction.

However, performance increase is not the stated goal of including MVR functions in the C∀ language. Even if there were no performance gains as a direct consequence of this feature, the *semantic* gains alone are important for the programmers developing or maintaining the code. If nothing else, the clear syntactic distinction between input and output arguments at the call site is helpful to the understanding of the intent of the code. To further illustrate this point, take for example the following fragment which invokes function foo:

```
foo(i1,i2,&o1,&o2);
```

It is unclear from this line whether the output arguments have to contain a sensible value to the function when it is invoked, or if this value is modified (if "transparent" pass-by-reference is possible, like in C++). These issues do not occur if the above code is stated as:

```
[o1,o2] = foo(i1,i2);
```

as it is plain now that the intent of o1 and o2 is to receive the values returned by a successful call to foo, and that these assignments do not interact with whatever happens during the evaluation of the right-hand-side. Also, it is now clear that i1 and i2 are not modified during the assignment or the evaluation of the right-hand-side (unless the language allows transparent pass-by-reference).

These are all useful facts that can be extracted directly from the syntax by the reader of the program, and the notation accounts nicely for one of the two possible usages of the values returned by MVR functions, as the right-hand-side of an assignment. This notation, first introduced to the C language in [26], requires the introduction of a new construct, a *tuple*. Tuples also fit in with the second usage of values from MVR functions, function composition.

The rest of the chapter describes in great detail the way tuples are included in C∀. It is organized as follows: a description of tuples and the operations on them is presented in the next section. The implementation of these operators is described next. Finally, additional notes on related work are briefly outlined.

4.2 C∀ Tuples

Recognizing the advantage of having MVR functions in C∀, its designers strived to find the abstraction that best included this feature and integrate it as seamlessly as possible in the overall fabric of the language. They found it in one of the most innovative aspects of its forerunner, KW-C.

KW-C [131] includes an abstraction of argument lists, a programming language device that is so commonplace that it is not often considered a construct in its own right, but a mere byproduct of the function call syntax. The rationale for identifying argument lists with tuples originally was that functions are often called with the same arguments, so giving the list a name made it easier on the programmer, and generated a possible compiler optimization. C∀'s tuples soon transcended its originally intended purpose by allowing the expression of a number of different concepts.

Tuples are ordered, fixed-size lists of possibly non-contiguous, heterogeneous elements. Such lists should be familiar to most programmers, as they appear in a number of contexts in imperative and functional languages: parameter and argument lists, array subscripting, fields of records, etc. In all these situations, tuples can be considered as a structuring device rather than the specification of a family of types. C∀ tuples and their concomitant programming constructs constitute a natural representation for several often-used programming devices and provide a powerful way of expressing programming ideas.

In C∀, the syntax of tuples is given by the grammar:

⟨*tuple expression*⟩ ::= '[' ⟨*tuple expression*⟩ [',' ⟨ *tuple expression* ⟩]* ']'
    | '[' ⟨*assignment expression* ⟩? [',' ⟨*assignment expression*⟩?]+ ']'

Square brackets, [], allow differentiating between tuples and expressions containing the C comma operator. Examples of tuples are (assuming the function application expressions contained therein return a single value):

```
[ 4, f() ]                     // 2 values
[ 7, ( f(), g() ) ]            // 2 values, comma operator
[ x + y, , 'a' ]               // 2 values, hole
[ int, double, int ]           // 3 types
[ 'a', ['b','c'], 'd' ]        // 4 values, nested tuple
```

Tuples can be arbitrarily nested. Not all forms of tuples are legal in all contexts where tuples are allowed, e.g., tuples with holes.

Tuples in KW-C, and therefore in C∀ are influenced by the set-based language SETL [45][1], which accounts for the sharp difference between this construct and constructs of the same name present in other programming languages, like ML, Python or Haskell. In particular, the individual elements in a C∀ tuple are not directly addressable, neither by name nor index nor offset (further exposing their potential non-contiguous nature). As well, a C∀ tuple does not model a sequence, so it is impossible to cycle through the contents of a tuple. Essentially, a C∀ tuple is largely a compile-time phenomenon, having little or no runtime presence. Therefore, it is wrong to equate C∀ tuples with tuples in other languages because the purposes of each are completely different. C∀ has different facilities and mechanisms to create the kinds of entities called tuples in other languages.

Tuples are not first class values in C∀. Their structure is also less strict than that of records, e.g., nested tuples are implicitly flattened. When passed to a function, tuples are implicitly opened to access their components, which are subsequently paired with the corresponding arguments, and when returned from functions a similar operation takes place. Functions do not return "a tuple of...", but multiple values of the corresponding types. The only exception is when tuples are used in contexts that require types, specifically in declarations (when declaring tuple variables). Tuples are best understood as a syntactical device, a shorthand notation that is expanded at compile-time, and that has little or no run-time manifestation. As such, their use does not enforce a particular memory layout, and in particular,

[1]SETL combines features suitable for symbolic programming with an imperative syntax and semantics.

does not guarantee that the components of a tuple occupy a contiguous region of memory. Some operations that are common to values, such as querying for the address are disallowed for tuples. Another consequence is that functions on tuples cannot be defined. Finally, a user cannot extend the built-in tuple operation set (described below). Essentially, tuples are used as a compile-time device to organize information.

4.2.1 Tuple Assignment

The structure of tuples is fluid. Although tuples are permitted to nest, they are immediately flattened when used, and their contents are implicitly extracted when required. In particular, in an assignment operation between tuples, i.e., an assignment that contains a tuple expression in its left-hand side, both operands to the assignment are implicitly flattened, and tuple variables are expanded to their definition. The components of both sides are then paired and individual "scalar" assignments are performed.

Holes in tuples introduce a more textured matching discipline, which is described below, but in general, tuple matching takes place between two flat lists of values. Once an assignment is performed, the left-hand side is conceptually restructured if required.

4.2.2 Multiple Assignment

Multiple assignment is the straightforward extension of simple assignment to tuples of the same size. It consists of a tuple of lvalues being assigned a tuple of expressions, taking the form:

$$[lvalue_1, lvalue_2, ..., lvalue_n] = [expr_1, expr_2, ...expr_n]$$

The left-hand side is a tuple of lvalues, which is a list of expressions each yielding an address, i.e., any data object that can appear on the left-hand side of a conventional assignment statement. Each *expr* appearing on the right-hand side of a multiple assignment statement is any standard arithmetic expression and its value is assigned to the corresponding lvalue on the left-hand side of the statement. Clearly, the types of the entities being assigned must be type compatible with the value of the expression.

The multiple assignment construct has parallel semantics, which permits a "swap" of the contents of variables to be written as:

```
[x,y] = [y,x]
```

(as in the case of argument lists, care must be taken when using side-effect expressions inside a tuple, since no particular order of evaluation is guaranteed by either C or C∀).

A special form of pattern matching takes place when "holes" appear in the left-hand tuple of the assignment operator:

```
[int x, int y, int z] foo();
[ret, , ] = foo();    // ignore last two values
[a,,c] = [x,y,z]      // ignore middle value
```

In both assignments, the rvalues in positions corresponding to the holes are ignored by the rest of the computation (and performed only for side effects).

If a function returns lvalues, holes can appear on the right-hand side of an assignment, as in:

```
[lvalue int x, lvalue int y, lvalue int z] bar();
bar() = [v1, , v3];
```

which results in the value associated with the second address remaining unchanged, while the results of the expressions v1 and v3 are assigned into the address specified by the first and third return values of the function bar. Note the keyword **lvalue**, a C∀-specific extension for a restricted pointer on which it is impossible to perform arithmetic and that is implicitly derefenced. **lvalue**s are similar to C++ references.

Mass Assignment

A convenient simplification of multiple assignment in C∀ is to assign a single value to a number of different variables, an operation called *mass assignment*, which has the form:

$$[lvalue_1, lvalue_2 ... lvalue_n] = expr;$$

where for all $lvalue_i$ provide the address of an object that is type-compatible with the type of $expr$.

As for multiple assignment, mass assignment uses parallel semantics, which means assignment is not equivalent to either C code fragments:

```
lvalue_1 = expr;
lvalue_2 = expr;
⋮
lvalue_n = expr;
```

or

```
lvalue_1 = lvalue_2 = ... = lvalue_n = expr;
```

The first fragment causes multiple evaluations of `expr`, which is, at the very least inefficient, and at worst, wrong, when an `expr` has side-effects. In the second code fragment, the value of `expr` is repeatedly casted into the types of lvalue_n, lvalue_$(n-1)$ and so on, which can cause loss of information along the way.

Parallel assignment semantics, ensures $expr$ is only evaluated once, precluding side-effect problems, and this value is assigned to each of the lvalues so that only the minimum typecasting takes place between lvalue_i and `expr`.

4.2.3 MVR functions in C∀

The introduction of tuples to C∀ permits the specification of functions returning multiple values that are consistent with SVR functions and, consequently, user's expectations. The result is a natural extension of C's syntax and style:

```
[int, int, int] gives3(int);
[x, y, z] = gives3(x);
```

Here, gives3 returns three values, which are assigned left to right into variables x, y and z. Or, if composing gives3 with another function:

```
takes3( gives3(w) );
```

where the multiple values generated by gives3 become the arguments of takes3.

A MVR function declaration may or may not associate names with the components of the return tuple:

```
[int, int, int] foo() { ... }    // unnamed return values
[int x, int y, int z] foo() { ... } // named return values
```

In the second form, the return-tuple component names become local variables in the function just like parameter names. This form introduces a similar facility to the short-lived gcc "named return values" extension[1]. There are several ways a MVR function can return a result:

```
[int x, int y, int z] foo() {
  [int, int, int] temp;
  // case 1:
  return temp;  // use of a tuple variable,
  // case 2:
  return [3,4,5];  // return ''tuple literal''
  // case 3:
  [x, y, z ] = [3,4,5];
  if( x == 4 ) return;
                    // ''fall-off'' the function (implicit return)
}
```

[1]Michael Tiemann, with help from Doug Lea, provided named return values in g++, circa 1989 [86].

Cases 1 and 2 directly return a tuple value. Case 3 indirectly returns a tuple value through the named tuple variables, i.e., a **return**; or fall-off the end of a function is rewritten to **return** [x,y,z];. The named case can help the compiler to optimize out unnecessary copying of temporaries from the function to the call site.

It can be seen that Mesa's (§4.1) design has been highly influential in this language extension.

4.2.4 Named parameters

Syntax imposes significance in the ordering of the parameters in a function that is not always warranted. This ordering has to be respected when calling the function. However, when using library functions, coming from a variety of sources, there is no hope for a universally respected convention regarding the order of arguments. Changing an interface is not always possible or even desirable, so a method of placing the sources of the parameters in the right position in the function call is required. This issue compromises the flexibility of function composition.

Recognizing this fact, C∀ includes *keyword parameters*. Keyword parameters [64] introduce an alternative ordering to the traditional *by-position* in parameter lists, by also adopting indexing *by name*, thereby rendering argument lists isomorphic under permutations. They also provide a dual for the named return values of an MVR function. That is, keywords can be used in an argument list to directly connect arguments to parameters, which is especially useful to rearrange tuples returned from an MVR function called as an argument. In C∀, having different names for parameters or return values in a function declaration (as a prototype) and later in its definition is considered an error.

Named parameters have been a part of programming languages since the early days of programming languages. Parameters are usually accessible by name within the function body[1], but, in most programming languages, not from the point of invocation. It is often inconvenient for a programmer to remember the order of a

[1]Although admittedly primitive programming environments like most Unix shells or the TEX typesetting environments use the position of a parameter to refer to it.

function's arguments, and accidental transpositions are not detected by the compiler, causing hard-to-find bugs. For a reader of the same code, things are not clear either, even when there is appropriate documentation or programming conventions that use (possibly temporary) variables named after the corresponding parameters, or that make clear what the purpose of the argument is.

Clearly in many contexts the order of parameters is either highly conventional (as for geometric coordinate systems) or immaterial (a function that is intensionally equivalent under a permutation of its arguments, as is the case, for example, with a binary commutative operator). In C∀, functions can be called with a mixture of positional and named arguments. While named keywords alone increase the expressiveness of a programming language, they are particularly useful when combined with other features, particularly *default parameters*, and composing MVR functions. Consider the following examples:

```
1 [int, int] foo();
2 void bar(int a, int b, int c);
3
4   bar(foo(), 3);
5   bar(3,foo());
6   bar([c,a]:foo(), b:3);
7   bar([a,]:foo(),2,3);
8   bar([,a]:foo(), 2,3);
```

In lines 4 and 5 above, the values being returned by function foo are matched with the arguments of bar based on their positions, in the usual fashion. However, it is unlikely that the writer of an MVR function knows the order the returned values are going to be required by other functions in a program. A *designated tuple* makes this knowledge unnecessary, as the order in which these arguments are to be matched with the parameters of the receiving function is explicitly specified. For example, in line 6 above, the results of foo are paired with arguments c and a respectively, whereas argument b is given by the integer 3. Furthermore, not every value from the MVR needs to be used. Any returned value can be discarded by the calling function, as illustrated by the last two lines in the example. In line 7, the first value from foo is paired with a, while the second is ignored; in line 8 the exact reverse is done.

Default values for parameters have been included in a great variety of programming languages. C∀ adopts C++ syntax for this facility, so default parameters follow each parameter's declaration in a function declaration, e.g.,

```
int foo(int x = 10, char abc = 'a');
```

Also adopting a widespread convention, C∀ requires all parameters that do not take default values (also known as *positional* parameters) to be listed first, followed by the ones with default values.

Technically, overloading and default parameters are redundant, since it is possible to achieve the effect of default parameters exclusively by means of overloading. This, however, requires a function declaration for each possible form of call, resulting in linear growth. As well, overloading cannot handle default arguments in the middle of a positional list, via a missing argument, such as:

```
p(1,/* default */,5);
```

The pattern-matching taking place between the function call and the function parameter list results in a rewritten call (and several assignment to temporaries), this time undesignated, that achieves the same effect. In brief, the matching process takes place by first constructing an ordered list of the parameter names of a function, and maintaining a pointer to the last bound parameter (initially, the first of the list). Upon an undesignated reference, the pointer moves forward to the next unbound parameter, which is then bound to the argument. When encountering a designated reference, the corresponding argument is bound, but the pointer does not move. At the end of the argument list, all unbound parameters take their value from their defaults if they exist. The "holes" in argument lists are matched positionally to the parameter list.

These pattern matching rules can be more formally described in terms of the well-known relational algebra operators projection (π), renaming(ρ), cross product ($\times$) and a nontraditional join ($\bowtie$) [42]. Under this interpretation, a function call is treated as a record operation. Each function call in the argument list is represented by its name, and it stands for its return value, which is always a (possibly one-) tuple; also, the argument-list comma delimiter stands for the cross product (concatenation) of tuples. The first step, the possible filtering of some of the values returned from a function, can now be expressed as a projection based on the position of the values in the return tuple. Furthermore, the rearrangement of arguments can be perceived now as a renaming followed by a (non-standard) join. For example, given the function declarations:

```
foo(int a,int b);
[int,int,int] bar();
```

the relational algebra formulation of the function call `foo( [,b,a]: bar() )` is:

$$foo \bowtie_{C\forall} \rho_{b,a}(\pi_{2,3}(bar))$$

where the second and third results from bar are selected (via the projection $\pi_{2,3}$), then renamed to b and a respectively ($\rho_{b,a}$), and finally joined ($\bowtie$, for C∀'s interpretation of a join) to the parameter list of foo.

It is easily noticed that the operation represented here as a join is not the traditional relational-algebra natural join, although its similarity is conspicuous.

These rules for matching arguments to parameters in function calls are also used to match initializers to fields when initializing aggregates, with two differences: the pointer always points to the last field initialized, whether designated or undesignated. Furthermore, a member can be referenced in an initializer list multiple times, either via designation or by being indicated by the current field pointer. Only the last value paired with the member takes effect. For function calls, repeated references to the same parameter via designation are considered errors. All these related operations are illustrated in figure 4.1.

If it so happens that after rearrangement the call still matches more that one function due to overloaded symbols, the *minimum conversion cost* [16] rule is used

$$foo = (a, b, c);$$
$$bar = (b, c, d);$$
$$foo \bowtie bar = (b, c)$$

(a) Natural Join

$$foo = (a, b, c);$$
$$bar = (b, c, d);$$
$$foo \bowtie_{left} bar = (a, b, c)$$

(b) Left Join

```
struct { int a, b, c } rec = { b:3,4; }; // {unitialized, 4,3}
```

(c) Record Initialization

```
int foo(int a, int b, int c = 10);
foo(b:3,4);   // foo(4,3,10);
```

(d) Argument list rearrengement

Figure 4.1: Name driven matching

to disambiguate the call. Essentially, this rule chooses the option that entails the least number of intermediate conversion (safe and unsafe) and specialization operations.

All these facilities together allows for a clean description of certain algorithms. For example, consider a function that takes a string and returns a permutation of it, "pivoted" around the center, that is, for the word `"overhang"` it returns `"hangover"`, for the word `"overturn"` it returns `"turnover"`, etc. A C∀ implementation of such a function, in terms of two auxiliary functions, split and concatenate (a multi-valued version of the Standard C library's strcat) is:

```
[char *s1, char *s2] split( char *s );
char *concatenate( char *s1, char *s2 );

char *mirror( char *s ) {
  return conatenate([s2,s1]: split(s));
}
```

4.2.5 Records and Tuples

Records (**struct**s in C parlance) are another instance of an ordering relationship unwittingly introduced by the language, this time among the fields. Since field selection in **struct** instances is done in terms of names, the existence of this position-based ordering is not really an issue, except for initializers, where the (textual) layout of the structure has to be mimicked by the initializing expression. C99 recognized this fact and introduced *designated initialization* (for **struct**s and **union**s), where each initializing expression is qualified with the name of the field it applies to. For example:

```
struct point {
  double x,y;
  int color;
}

struct point p1 = { color=RED, 3.0, 4.0 };
```

C∀ adopts this innovation with a slightly modified syntax (the assignment symbol is replaced by a colon), and naturally extends the definition of designators to include name tuples. For example, an instance of the type **struct** point in the above example:

```
struct point pcfa = { color: RED, [y,x]:3.0 }
```

Tuple assignment and function calls can also be used when selecting multiple fields of a structure, since C∀ allows the field selection operator to take a name tuple to refer to multiple members of a **struct** instance:

```
struct st {
  int x,y;
  double z;
} s1, *s2;

s1.[x,y] = [2,3];   // multiple assignment
s2->[y,x] = 5;      // mass assignment
```

4.3 Implementation

Till [131] implemented a first-approximation of most of the operations described above (barring MVR function composition) for KW-C, an extension of the C programming language. The C∀ language, as described in [16] enriches the C language with parametric polymorphism and overloading, and the type resolution algorithm accounts for MVR functions. However, this work did not include tuples and their extended assignment forms, nor did it account for the code generation of this forms or MVR functions. The remainder of this section fills in these gaps. First, extensions to the type resolution algorithm mentioned above are described to account for various constructs resulting from the introduction of tuples. Finally, the code generation algorithm for these forms is presented.

4.3.1 Tuple expression analysis

Traditional overload resolution algorithms rely on the correspondence of the number and position of arguments in a function call with the number and position of the formals in a function definition. The not-so-traditional Baker-Ditchfield-Bilson[16] overload resolution algorithm also consider functions returning multiple values, but still assume positional correspondence. When named parameters are thrown into the mix, this assumption must be discarded, and further generalization of the algorithm is needed.

C∀'s overload resolution algorithm receives an untyped expression tree and it returns a typed tree, where each of the function calls therein resolved to a unique function in the program. The whole expression tree is then uniquely interpreted. This resolution process takes into account not only the number and type of the arguments to the functions, but the types of the returned values as well. It also accounts for functions returning multiple values and C's implicit type conversions. A full account of the algorithm is given by Bilson [16]. Suffice it to say here that it works in a bottom-up fashion, keeping track of every possible interpretation for each subexpressions and picking from among them once information on the context becomes available. A conversion-cost-based tie-breaking scheme is used to disambiguate calls. For tuples, this policy changes somewhat. The assignment that is finally performed is the *narrowest type* that applies to all the tuple's components. This policy ensures that expressions are evaluated only once, but may result in different effects than the pairwise macro-expansion interpretation. In fact, it might be the case that a tuple assignment is termed "invalid" by the system, even if the pair-wise expanded form has a valid interpretation. Consider, for example, the case:

```
double foo();
int *foo();
double d;
int *pi;

d = foo(); pi = foo(); // fine, 2 calls
[ d, pi ] = foo(); // invalid assignment (no narrowest type), 1 call
```

4.3.2 Tuple code generation

Multiple assignment

Once expressions have been matched on both sides of an assignment, temporaries are generated for each pair of left- and right-hand values. The former consist of variables of pointer type that take the address of the left-hand operands of the assignments, whereas the latter store the value of the right-hand operands. Although this strategy might seem like a superfluous generation of temporaries, it is necessary to ensure that expressions on either side are evaluated only once. The result can be seen in figures 4.2 and 4.3.

Mass assignment

A temporary is generated for the right hand side, and as many temporaries as are required for the left hand side. This allows the right-hand-side expression to be executed only once (with the corresponding side effects executed only once). The result is illustrated in figures 4.4 and 4.5.

MVR functions

As outlined in section 4.1, there are several options for the simulation of the effect of MVR functions in C[1]. Consider the C∀ program:

```
[int, int] divmod(int q, int d)
{
  return [ q div d, q % d ];
}

void display_pair(int, int);

display_pair(divmod(a,b)); // use
```

A first approach is based on packing and unpacking of structures, which is the

[1]There are even more than the ones illustrated here. For example, the use of coroutines, which generates code too complex and inefficient to be seriously considered.

```
[i,y[i],z] = [a + b,i,3];
```

(a) C∀ version

```
{
    int *__tpl_lhs_4;
    int __tpl_rhs_5;
    int *__tpl_lhs_2;
    int __tpl_rhs_3;
    int *__tpl_lhs_0;
    int __tpl_rhs_1;
    (__tpl_lhs_0=(&__i__i));
    (__tpl_rhs_1=(__a__i+__b__i));
    (__tpl_lhs_2=(&__y__A0i[((long int )__i__i)]));
    (__tpl_rhs_3=__i__i);
    (__tpl_lhs_4=(&__z__i));
    (__tpl_rhs_5=3);
    ((*__tpl_lhs_0)=__tpl_rhs_1);
    ((*__tpl_lhs_2)=__tpl_rhs_3);
    ((*__tpl_lhs_4)=__tpl_rhs_5);
}
```

(b) C translation

Figure 4.2: Code generation for a multiple assignment statement

```
[x,y] = [y,x];
```

(a) C∀ version

```
{
    int *__tpl_lhs_2;
    int __tpl_rhs_3;
    int *__tpl_lhs_0;
    int __tpl_rhs_1;
    (__tpl_lhs_0=(&__x__i));
    (__tpl_rhs_1=__y__i);
    (__tpl_lhs_2=(&__y__i));
    (__tpl_rhs_3=__x__i);
    ((*__tpl_lhs_0)=__tpl_rhs_1);
    ((*__tpl_lhs_2)=__tpl_rhs_3);
}
```

(b) C translation

Figure 4.3: Code generation for “swap” statement

```
[i,y[i],z] = foo();
```

(a) C∀ version

```
{
    int *__tpl_lhs_4;
    int *__tpl_lhs_2;
    int *__tpl_lhs_0;
    int __tpl_rhs;
    (__tpl_lhs_0=(&__i__i));
    (__tpl_rhs=__foo__Fi__());
    (__tpl_lhs_2=(&__y__A0i[((long int )__i__i)]));
    (__tpl_lhs_4=(&__z__i));
    ((*__tpl_lhs_0)=__tpl_rhs);
    ((*__tpl_lhs_2)=__tpl_rhs);
    ((*__tpl_lhs_4)=__tpl_rhs);
}
```

(b) C translation

Figure 4.4: Generated code for a mass assignment statement

```
[x,y] = a + b;
```

(a) C∀ version

```
{
    int *__tpl_lhs_2;
    int *__tpl_lhs_0;
    int __tpl_rhs_1;
    (__tpl_rhs_1=(__a__i+__b__i));
    (__tpl_lhs_0=(&__x__i));
    (__tpl_lhs_2=(&__y__i));
    ((*__tpl_lhs_0)=__tpl_rhs_1);
    ((*__tpl_lhs_2)=__tpl_rhs_1);
}
```

(b) C translation

Figure 4.5: Generated code for a mass assignment statement

technique KW-C adopted in its translator:

```
struct __divmod__Ret { int x; int y; } divmod( int q, int d ){
  struct __divmod__Ret ret;
  ret.x = q + d;
  ret.y = q * d;

  return ret;
}

/* use */
struct __divmod__Ret rec = divmod( a, b );
display_pair(rec.x, rec.y);
```

CPS, as described above, could also be used. Any of these approaches would profit by additional information at the call and within the function, namely how many and/or which return values are actually used at the call site. Such information can be utilized to optimize the call, since the callee need not compute unused results. Since C's arguments are unmoded, and can be aliases of one another, an optimizer for these expressions is non-trivial.

The current incarnation of the C∀ translator rewrites MVR functions to take extra by-reference arguments. It also rewrites every **return** statement in the function body.

```
[ int a, int b ] gives_two();

gives_two();
[x,y] = gives_two();
```

generates:

```
void gives_two(int *a, int *b );

{
  *__tup_x = &x;
  *__tup_y = &y;
  gives_two(__tup_x, __tup_y);
}
```

The case of composition with tuple designation is more interesting. This process takes place in stages: conceptually, an auxiliary multiple assignment statement is generated, taking into account all the potential rearrangement of arguments. The following C∀ code

```
  void takes_two(int a, int b);
  takes_two([b,a]:gives_two());
```

is transformed, in stages, to:

```
[__t1, __t2] = gives_two()
takes_two(__t2, __t1);
```

and then (C) code for it is emitted as output, generating the following:

```
  {
    {
      *__temp_t1 = &__t1;
      *__temp_t2 = &__t2;
      gives_two(__temp_t1, __temp_t2);
    }
    /* rearrange arguments */
    takes_two(__t2, __t1);
  }
```

4.4 Related work

Jaakko Järvi's `tuple` library [72] (now part of the `boost` [19] library suite) makes use of the template facilities in C++ and the so-called type lists [7] to add MVR functions to C++.

Matrix-manipulation languages like APL and Matlab allow for a wide range of modification of the form of its argument list via vector multiplication and matrix transposition. These operations, however, are not generalizable to heterogeneous lists.

Rearrangement of argument subexpressions via designators is a consequence of having keyword or named parameters in the language. It is surprising that, given the formal similarity of this facility with records, features of this nature have not been more widely studied. Two extensions to the λ-calculus that take into account record-like function calls and modify the substitution rules accordingly are Garrigue et al. "label-selective" λ-calculus [59] and Laurent Dami's λ-N calculus [41]. The latter is especially interesting as it uses designated argument lists as the basis for a record- and object-calculi for general software composition.

Similar behaviour to C$\forall$'s designated call can be achieved, in a much limited form, in Haskell via the library functions curry, uncurry and, in particular flip, which is a combinator that returns a function with its two arguments reversed. These facilities rely on the presence of closures, currying, and partial application (slices) in the language. Although these features make it possible for (positional) argument rearrangement to be written as a library, the library approach does not scale to handle tuples of greater length. It is worth pointing out that, in contrast with C$\forall$'s approach, Haskell modifies the function to be applied (flip returns a function) rather than the argument list.

Languages that use similar function-modifying mechanisms, like Lisp, Scheme, and, most notably, Dylan macros can also be used to achieve functionality similar to C$\forall$'s designated calls, in the same sense that the Haskell approach does. If a non-standard Lisp with reflective extensions is used, modification of the argument list is possible. Such an approach, however, is likely to introduce a sublanguage for characterizing the rearrangement similar to C$\forall$'s name tuples. Either way, these options require the presence of a very powerful and deeply embedded macroprocessor.

The C++ and Haskell approaches are inherently limited by having to hard-code the length of the tuple. This is derived from limitations inherent to their type systems. More powerful type systems, that include dependent types [141] are needed. Currently, there are very few languages that include this facility. An example is Lennart Augustsson's Haskell-inspired Cayenne [13]. Cayenne allows the user to write functions over tuples of any length (the most common example is an n-ary zip function). Tuple rearrangement, however, requires more complicated annotations on the tuple than just the length.

Chapter 5

Attributes

A program is often perceived as the description of a process in terms of a computational model. This view focuses on the programmer directing a computer to the solution of a problem. Higher-level programming languages enable a more expressive description of a solution, but also add content to the program, and participants in the interaction. A program can now be viewed as a message, whose content is more than just a computational mechanism.

For example, there are fragments in the program text that are directed to programmers rather than the computer, such as **const**, access qualifiers and other annotations, e.g., comments and variable names, that indicate how objects in the program are meant to be used. Naming conventions often make up for features missing in a language. For example, to address the lack of modules, identifiers in C are often prefixed with a library name; also, type variables in C++ STL code are named after the concept they model [58, 14] in an attempt to compensate for a lack of features to constrain generic types, like C∀'s contexts or Haskell's type classes. Lack of contexts or type classes (and functional dependencies [75]) also forces programmers to express relationships using naming conventions, like that between type **int** and its minimum and maximum values by identifiers INT_MAX, INT_MIN. In fact, it seems that carefully named identifiers are often used whenever a relationship between an object and other parts of the program cannot be expressed by linguistic means.

Other program fragments address the *implementation* of the computational model, directing the process of code generation, aiding the linking phase, or enabling the generation of runtime checks by manipulating a host of options and features, often in the form of "pragmatic comments" or *pragmas*. Pragmas are often given special syntax in the language, but this is not always the case. For example, C's type qualifiers **register** or **inline** are truly pragmas, since they supply information to the code generation mechanism. Having a non-*ad hoc* syntax for attributes allows for clearly-marked direct-communication between the programmer and the compiler. It also provides a unified interface to this kind of communication. Some compilers, like `gcc` extend the language to allow for richer pragma syntax in the form of *attributes*, which interface with the compilation/optimization process. The programmer can annotate only certain objects (functions, variables or types), which are called *targets*, with attributes. `gcc` attributes cannot be extended and only direct the compiler's translation.

However, it is possible to generalize attributes further. Ada, for example, uses attributes to communicate information both *to* and *from* the translation system and the programmer. In Ada, targets of attributes include program units (modules), labels, types, etc. Attributes contain information about targets such as string representation, version, low-level representation details (e.g., size and alignment of types), and relationships between entities, e.g., the base type of an access or constrained type. The list of attributes, although large, is fixed and determined by the language specification [71]. Compiler implementors may extend it, but not application programmers. Despite this limitation, programmers are allowed some degree of flexibility, since they can override the behaviour of certain attributes, specifically marked in the standard as *specifiable*, as long as the attribute's interface is respected, i.e., its target, extra parameters and return type.

Languages descending from Ada, like VHDL (VHSIC Hardware Description Language), generalize this rule by allowing programmers to declare their own attributes thereby providing a more general code annotation mechanism. Targets, however, are a restricted and fixed set of entities in the language. Attributes in Ada and VHDL can be queried for their value by an *attribute expression*, which consists of a reference to the target suffixed with the attribute name and extra parameters, if needed. Ada attributes provide a hook for extensibility in related development tools, like compilation managers and compilers, which take advantage of the attribute syntax to enable enhanced communication with the programmer.

Code annotation is also provided in C#, with a mechanism also called attributes. C# attributes are patterned after those in the DCOM and CORBA interfacing mechanisms. As in VHDL, attributes can be extended (user-defined attributes are known as *custom attributes*). The attributes predefined by the language furnish the programmer with the possibility to configure a great number of code generation features that are hardcoded in most programming languages, e.g., marshalling policies for distributed communication, memory layout, etc. Targets of attributes include all static language constructs. At runtime, targets can be queried for the value of their associated attribute values, which are stored is a special "metadata" table. C# attributes are extensively used by compilers, validation tools and execution environments.

Viewing the program as a message, and having multiple interested recipients (compiler, development tools, runtime), it is fair to ask whether programmers could profit from programmatic access to the source code, annotated or otherwise. Programs that can access their own source code have long been thought of as mere curiosities (e.g., the so-called *quines*, programs that print their own listing). However, *metaprogramming*, i.e., the ability to manipulate source code as data coupled with a means to automatically generate code, allows for extremely generic code that is customized at compile- or even run-time to great gain in adaptability and performance. Metaprograms, or programs that specify how other programs should be generated, are the subject of study in the field of Generative Programming [40]. As a side effect, when a program can access its own source code, an additional mechanism to query code annotations is reduced.

When access to some aspect of the source code is required by a programmer, it is common to resort to a hand-duplicated version of the program fragment (e.g., C++'s "smart enumerations" [126] expect the programmer to provide the enumeration constants twice: one in the **enum** and another as a string), or to circuitously deduce it from the program (e.g., the `type_traits` C++ template library takes advantage of the Turing-completeness of the template instantiation mechanism to find out type properties by observing the effects of the overloading resolution process). Both these solutions involve either tedious and error-prone programming practices, very detailed low-level knowledge of the inner workings of a language mechanism or compiler, or, in the case of C++ template metaprogramming techniques, the use of a very powerful mechanism for other purpose than it was intended. It should be possible for the language to allow these problems to be expressed directly, given that the information is already available within the program text or the compiler's symbol table. Furthermore, the compiler is privy to information about the execution environment of a program. Access to this information can significantly add to the flexibility and adaptability of a program.

A mechanism that provides a unified access to the source code and certain aspects of the execution environment is that of *reflection*, which is described in the following section. Reflection is not the only way to enrich program entities with annotations accessible programmatically, and some of the alternatives are discussed in the following section, in particular with regards to types. C∀'s attribute mechanism is then presented, consisting of a combination of already in-place mechanisms like overloading and new attribute mechanisms. Finally, some related work is mentioned.

5.1 Reflection

The original meaning of *computational reflection* is the ability of a program to inspect and manipulate itself at the source-code level. Mainstream programming languages that incorporate this concept, like C# and Java, extend the scope of the reflection mechanism to include the program state and its execution environment. This section concentrates on the aspect of reflection that relates to program code. Program-code reflection usually involves two operations: *reification*, which translates program fragments to data, and *splicing*, which does the reverse.

Full reflection is an ambitious goal, and few programming languages provide it. However, even a limited set of reflective features is useful. Reification could be limited to certain kinds of language constructs, as could be the operations on these representations, or the form in which they are spliced back onto the running program. Depending on the operations allowed over code-fragment representations, a classification of reflective systems is possible [44]. If the reified data is available only for querying, the system is said to provide *introspection*; if write access to the representation is allowed, the system has *linguistic* or *structural reflection*; and finally, if an updatable program representation includes aspects of the semantics of the programming language, e.g., evaluation order of arguments, the system has *behavioural* or *computational reflection*. In systems with linguistic or computational reflection, a reified program representation can be modified and spliced into the running program, allowing for the possibility of run-time code generation. However, this possibility is not considered further in this work.

Computational reflection was first introduced in the context of Lisp [117], a language whose uniform representation of code and data makes the problem of reification/splicing that of (quasi)quoting/evaluating. Other languages with similar uniformity, like Prolog, soon started benefitting from metaprogramming techniques, and it has become part of the Prolog lore to modify the usual search strategy, implement explainers or debuggers, and similar applications by replacing the standard evaluation function using "metacircular interpreters".

Interpreted, dynamically-typed programming languages can benefit from reflection very naturally, since they can encapsulate the interpreter itself as a function taking a string argument containing a code fragment and interpret it. Results and effects (e.g., new object creation) contained in the code fragment can all be readily incorporated into the running process. Interpreted, object-oriented dynamic languages take a more systematic approach to program representation. Usually, a language-provided framework, known as a metaobject protocol (MOP) [79], provides an object-oriented interface to the inner workings of the runtime and program representation. Through the MOP objects, the program can be queried, navigated and manipulated. The resulting flexibility of this design accounts for much of the adaptability of programs written in languages like CLOS or Smalltalk.

In statically-typed languages, the problem of reflection is more complicated, and only recently the first complete implementation of a metacircular interpreter for a Turing-complete, statically-typed language was presented by Läufer and Odersky

[85]. Due to the current prominence of applications that require security, adaptability, persistence, and a host of other characteristics that reflection enables, much interest in reflection for statically-typed programming languages has arisen (see, for example, Stemple et al. [120]).

Perhaps the most complete interface to the state of the execution and program representation in a statically-typed language is that of the SML dialect of ML. The SML of New Jersey interactive compiler provides metaprogramming and separate compilation by externalizing various internal compiler representations and processes to make them available to ML programs, a technique rendering a *visible* implementation of the compiler and supporting runtime libraries [11]. Although not everything in a ML program can be manipulated, the available externalized entities coupled with code annotations have been sufficient to enable interesting applications, such as the SourceGroup library [109], that plays the rôle of `make` in a C/Unix environment, Blume's foreign function interface [18], or Tolmach's debugger [132], which works by automatically instrumenting source-code and provides traditional breakpoint/watch mechanisms, reverse execution, checkpointing and replay.

However, this sort of uniform interface and powerful interaction with the state of the program is very rare. Currently, most mainstream statically-typed languages like Java or C# include a limited, yet useful set of reflective features. In this work, I focus on the introspection aspect of reflection, comprising the ability to programmatically examine (not modify) various program objects, especially types. Code that take advantage of this ability is robust to changes in the definition of types.

5.2 Introspection

As mentioned above, introspection is the read-only access to the program representation. A minimal set of introspective capabilities are usually included in most languages. For example, introspective features have been part of C since its inception, via the C preprocessor[1]. Programmers have programmatic access to the current file name and line number via the __FILE__ and __LINE__ macros respectively. Aspects of the compilation environment can also be captured from the value of environmental macros, such as __STDC__, a boolean that determines whether the compiler is Standard-C compliant, or __DATE__, that provides the system's date. Due to the nature of the C preprocessor, certain information is not accessible through macros. In particular, the preprocessor is blind to function names and any scope considerations, which forced the C99 standard to extend the introspective information in a different way by introducing the predefined identifier __func__, which is declared implicitly (if used) within a function as:

```
static const char __func__[] = "function-name";
```

where *function-name* is the name of the function the identifier is used in.

Other introspective information the C compiler makes available is accessible through a somewhat inconsistent and disconnected interface, consisting of the operators address-of (&), **sizeof** and **offsetof**, and **__alignof__**, **typeof**, and address-of-label (&&) as gcc extensions. Access to the environment is also done in C via operating-system calls.

Introspection of types is particularly useful, as it allows programmatic access to structural properties that are important for transmission or manipulation of data in the program. The only information C discloses on a type is lower-level representation information, via **sizeof**, **offsetof**, etc. In C++, better introspection facilities can be built via conventions and abstraction mechanisms, which allow for the compile-time annotation of types.

A C++ idiom, known as *trait classes* [95], is used to annotate types at compile time. A trait class is a compile-time device that permits the annotation of a type with information such as associated values and types. A trait is a template class that encodes values and types associated with its type parameter. This idiom relies on *partial template specialization*, which permits the template creator to specify a

[1] These capabilities constitute one of the reasons why the C preprocessor is so difficult to eradicate, despite its myriad detractors.

```
template <typename T> struct my_type_traits {
  static string get_string() { return "unknown"; }
};

template <typename T> struct my_type_traits<T *> {
  static string get_string() {
    return "pointer to " + my_type_traits<T>::get_stringrep();
  }
};

template <> struct my_type_traits<int> {
  static string get_string() { return "int"; }
};

  cerr << my_type_traits<int>::get_string() << endl; // "int"
  cerr << my_type_traits<int *>::get_string() << endl; // "pointer to int"
  cerr << my_type_traits<char *>::get_string() << endl;// "pointer to unknown"
```

Figure 5.1: Example of C++ trait classes

"primary template" for general types, more specialized versions for types of certain structure and completely specialized versions for specific types. With this mechanism, C++ provides a minimal pattern language, based on type constructors (not to be confused with *class* constructors). An example of a trait class is shown in figure 5.1.

There are several libraries that encapsulate information about a type in trait classes. Among these, the best known is `type_traits` by Maddock et al. [89]. The information contained in these classes includes inheritance details (e.g., whether a class is derived from a particular base class), and type ordering (whether there is an implicit conversion from a type to another). Providing this information as a library has the advantages that no language or compiler modifications are necessary, and that extensibility is guaranteed. However, the information accessible in this way is limited. Proposals for linguistic support for type information have been submitted to the C++ Standardization Committee [121], as have proposals for more general compile-time reflection facilities [134, 73].

For programmatically-accessible annotation of program entities, types, in particular a C programmer must resort to the creation of a runtime type system, like that of the GNOME GObject library [128], a framework that provides aspects of the C type system at runtime. In particular, each (registered) type in the system is uniquely identified by a number, which then allows for the association of arbitrary information via a dictionary.

C++'s runtime type information (RTTI) facility allows for the annotation of types, both built-in and user-defined [123]. The compiler generates some information per type (including a string representation of the type name and a suitable equality test between type data-objects) and stores it in instances of class type_info, which are (read-) accessible via the function typeid. typeid takes the name of a type or an expression, and returns the corresponding type_info object. The design of type_info allows for extending type information, e.g., by building dictionaries indexed by type name. This form of introspection, although useful, is limited. For example, an instance of a class cannot be queried at runtime for its public methods.

Compared to C and Ada, Java includes powerful introspection capabilities, that allow for the easy development of meta-language facilities such as class browsers and interactive program generators. More interestingly, they provide for object *serialization* in the form of a library, where serialization is the process of writing or reading an object to or from a persistent storage medium, such as a disk file.

5.3 C∀ attributes

Because C∀ is based on C, it can use C's previously mentioned mechanisms to access information about a program and its execution environment. Interestingly, C's ad hoc introspection mechanism for types (e.g., INT_MAX) can be better expressed through C∀'s powerful type system by using programming idioms and conventions. This solution, however, does not apply to all introspection needs. To provide for these additional needs, I have designed and implemented a new mechanism, inspired by Ada attributes that integrates well into the existing syntax and semantics of C∀. This mechanism unifies several ad hoc mechanisms in C, its preprocessor and runtime system. Both programming conventions and the new mechanisms for introspection are described in this section.

5.3.1 Idioms and conventions

All the introspection capabilities described above for C also apply in C∀. Moreover, compile-time type annotation is also available, as in C++, via the overloading resolution mechanism, e.g.:

```
forall (type T) string get_type_string(T t) { return "unknown"; }
forall (type T ) string get_type_string(T *t) {
  string ret = "pointer to ";
  return ret + get_type_string(*t);
}
string get_type_string(int i) { return "int"; }
```

Since the C∀ type system allows for overloading of variables, this technique is also useful to designate special values associated with a type. The exact relation of the value with the type is given by an appropriately descriptive variable name. Consider, for example, types with a *maximum* value, like those shown below.

```
int max = INT_MAX;
float max = FLOAT_MAX;

forall( type T | { T max; T end; _Bool ?!=?(T, T);
                   _Bool ?<?(T, T); _Bool ?>=?(T, T);
                   T >>(istream is);} )
```

```
T find_min(T lower_bound) {
  /* read Ts from an input stream, find min from read values
     greater than lower_bound */
  T cur = max;
  while( cur != end ) {
    T temp;
    is >> temp;
    if ( temp < cur && temp >= lower_bound )
       cur = temp;
  }
  return cur;
}
find_min(7); // uses max = INT_MAX
find_min(−3.14); // uses max = FLOAT_MAX
```

Although naming conventions allow for certain forms of type annotation, it is not useful for the association of arbitrary data with types. Therefore, a specialized mechanism is needed.

5.3.2 Attribute mechanism

There is interesting information about the program that is inaccessible through the overloading resolution process or any other mechanisms within the language. To gain access to it, a new mechanism is necessary, that of *attributes*.

Rather than Ada's suffix notation, C∀ adopts a prefix function-like notation, which makes clear that the association is done on types, while reusing a familiar syntax. C∀'s attributes are regular C identifiers prefixed with the character '@'. Their use in expressions follows the grammar in figure 5.2.

Attributes allow for the provision of a minimal program representation through which limited introspective capabilities are provided. Most reflective systems assure consistency between the reified program representation and the actual program by "opening" the compiler and providing hooks into the intermediate representations. C∀ takes an alternative approach, which can be described as lightweight [49]: the

$\langle \textit{attribute\_expression} \rangle$ ::= $\langle \textit{attribute\_identifer} \rangle$
    | $\langle \textit{attribute\_identifer} \rangle$ '(' $\langle \textit{type\_name} \rangle$ ')'
    | $\langle \textit{attribute\_identifer} \rangle$ '(' [$\langle \textit{assignment\_expression} \rangle$], ')'

Figure 5.2: C∀ attributes grammar

translator generates language objects that provide access to the compiler's program representation. This strategy removes the tension between having a representation that is convenient to manipulate by the programmer and at the same time efficient to implement by the system. On the other hand, since the representation of the program is created by the compiler from what the compiler itself is using, consistency is also guaranteed. Consider the following example:

```
enum work_week { MON, TUE, WED, THU, FRI, };

for( enum work_week day = @enum_first(work_week);
     day <= @enum_last(work_week); day= @enum_next(day) ) {
  printf("Opening hour in day %s: %d\n", @enum_name(day), op_hour[day]);
}
```

becomes:

```
enum work_week { MON, TUE, WED, THU, FRI, };

char *__enum_name( enum work_week e ) {
  switch( e ) {
  case MON:
    return "MON";
...
  }
}
```

```
enum work_week__enum_next( enum work_week e ) {
  switch( e ) {
  case MON:
    return TUE;
...
  }
}
for( enum work_week day = MON; day <= FRI; day = __enum_next(day) ) {
  printf("Opening hour in day %s: %d\n", __enum_name(day), op_hour[day]);
}
```

Notice that certain built-in attributes are compile-time constants, and are inlined. This strategy allows for declarations of the form:

```
int hours_per_day[ @enum_size( enum work_week ) ];
```

A listing of C∀'s expression-returning built-in attributes is presented in table 5.1. The attributes `@enum_next()` and `@enum_previous()` allow enumerated values mapping to a non-contiguous set of integers. The introspective capabilities into structures are fairly limited, and only permit the enumeration of the field names and number of fields. These attributes are provided as interesting examples, as it is clear that the list can grow significantly.

Not all built-in attributes are related to types, for example, the `@func` attribute, provides the same kind of information as C99's `__func__` identifier, in providing the name of the current function. This piece of information is useful, for instance, to keep track of the source of an exception:

```
terminate mkIO_exception(@func());
```

As a result of the implementation of polymorphic calls, the C∀ inner machinery maintains a runtime *type descriptor*, which contains such information as the size and alignment of a particular type. It also contains values and functions that are common to all types, such as a string representation (like that of of C++ `type_info`), and an assignment function. This information is valuable in several situations.

Prototype	Meaning
`char *@func`(void)	Name of current function
`unsigned int @enum_size`(enum_type)	enumeration size
`char *@enum_name`(enum_type elm)	string representation of enum constant
`enum_type @enum_previous`(enum_type elm)	previous value in enumeration
`enum_type @enum_next(enum_type elm)`	next value in enumeration
`enum_type @enum_first(enum_type)`	Takes enum name or value and returns first constant in enum
`enum_type @enum_last(enum_type)`	Takes enum name or value and returns last constant in enum
`char ** @struct_fieldnames(struct_type)`	name of fields in struct
`unsigned int @struct_numfields(struct_type)`	Number of fields in struct
`char * @typename(type)`	string representation of a type

Table 5.1: Built-in attributes in C∀

Consider a programmer wishing to find out what type is instantiated by the system for a particular polymorphic call:

```
forall( type T ) void polyfun( T t ) {
  printf("Type T was instantiated in function %s as: %s", @func(), @typename(T));
  // ...
}
```

A programmer can override the behaviour of a generated function by providing their own implementation, as long as the interface is respected. This information, combined with richer metaprogramming facilities, would allow programmers to control or specify the automatic generation of code (*à la* macros or templates). Although C∀ is currently nowhere near this stage, the attribute mechanism is a step in the right direction.

5.4 Related Work

Current programming tasks require flexibility (e.g., localization) and interaction with external data and computational sources. These requirements suggest that it would be beneficial for the programmer to have a single form to provide all auxiliary information on program elements [138]. This information can then be used by development tools, development tools, deployment tools (e.g., *stub generators*), or run-time libraries. Already incorporated in C# in the form of attributes, metadata facilities have been proposed for the upcoming version of the Java programming language [17], and furnish Java with a means of associating arbitrary attribute information with particular classes/interfaces/methods/fields.

The integration of annotations with the C programming language has not been widely explored. The intensional programming language Intensional C `icc` [74] allows the annotation of identifiers (functions and variables) with *versioning* specifications, and includes algorithms to best match versions of software components. This information is accessible at runtime via a new keyword, "vswitch".

The automatic generation of functions that access complicated data objects has been studied by Sheard [115] in the case of recursive types, and by Grossman [62] and Chuang et al.[29]. Closer to the approach presented here is the work by Willink et al. [137], which relies on a deeply embedded object-oriented preprocessor in C++, enriched with meta-functions and meta-variables.

`libpdel` [2] is a library that allows a user to describe a C type in a homogeneous structure, and then works on this structure to provide marshalling into XML-RPC and other protocols. It also allows for introspection of field names. The Gnome GObject [128], in addition to these features, develops a complete dynamic-type system for C.

Chapter 6

C language compatibility

One of the C∀ programming language design objectives is to be backwards compatible with C, as the latter is defined in its Standard [6]. This goal was met to a large extent, but due to the new constructs introduced in C∀, and a desire to fix problems in C, a small number of incompatibilities exist. Among these, the most obvious are the new keywords and the modification to the behaviour of the multibranch selection-statement **switch** (section 3.2). Because of these changes, C∀ marks as invalid some legal C programs. This chapter is a first approximation to estimate the number of C programs that might be in this situation. To this end, a large body of code is searched for instances of the prognosticated incompatibilities. The machinery set in place to carry out this task also allows, as a side benefit, an estimation of the usefulness of certain C∀ extensions.

6.1 Experimental setup

This chapter describes an experiment consisting of scanning a corpus of valid C code for code fragments that have been made invalid in C∀, and code fragments that could benefit from new features of C∀. C∀'s incompatibilities arise from the introduction of new keywords and changes to control statements, in particular the change to the behaviour of the **switch** statement (§3.2). The first language change is unavoidable, and designers of every language that extends another must choose new keywords carefully, so as to cause the least disruption in existing code. The hypothesis of the experiment is that C∀ keywords seldom conflict with existing C variable names and other identifiers. As for the changes in the **switch** construct, I maintain that changes in C∀ codify existing best practices, already respected by programmers. Evindence to support these contentions is presented, giving credence to the claim that few conflicts should occur in legacy code.

The patterns searched for in the corpus of C code are:

Identifier clashes If a C program includes among its identifiers (variables, functions, formal parameters, etc.) uses of C∀ keywords, it is not valid C∀.

Changes to the switch statement C∀ has disallowed statements in between the **switch** and first **case**, which prevents usage such as Duff's device. It also modifies the behaviour of declarations in this position by guaranteeing that initialization is performed. How limited the impact of these C∀ changes is can be ascertained by observing the following properties of the corpus:

- The number of **switch**es. The effects of changes to a language construct are necessarily more localized if the construct is used sparingly.
- Whether **switch**es are intertwined with other control structures. Any code that exhibits this property would be disallowed in C∀.
- Little or no code is placed between the **switch** and the first **case**. Any code that exhibits this property could have its meaning changed in C∀.

Fall-through case clauses The usefulness of certain C∀ extensions can be estimated by how often a C pattern (idiom) is used that can be replaced by a simpler C∀ mechanism. There are two main idioms for using fall-through.

The first one is for cascading a series of options that overlap, e.g.:

```
switch ( argc ) {
  case 3:
    // open outfile file
    // fall through to handle input file
  case 2:
    // open input file
    break;
  default:
    // print usage message
}
```

The second is to concatenate **case**s in such a way as to make up for C's lack of lists and ranges as **case** guards. By examining the use of fall-through **case** clauses, it is possible to determine when this idiom can be replaced by C∀'s **case** ranges or lists. This, of course, is just an initial approximation, as new idioms and styles of programming may arise from having more complicated **case** guards, and combining it with other facilities in the language.

Subsequent sections describe the corpus of code examined, how this population is sampled, the tools and techniques by which the sample is analysed and, finally, the results of the analysis. These results provide the evidence to conclude that the modifications made in C∀ do not affect a large amount of existing C code. This outcome suggests that the restrictions C∀ imposes are consistent with existing coding conventions; as a result, programmers who observe these conventions will not encounter the incompatibilities.

6.2 Corpus

The code that constitutes the corpus over which the search is conducted is obtained from the "open source" movement. Many of these programs, freely available in source form, are intended to be production-quality, to be used in day-to-day operations. Most of them have been written adhering to some coding standard and have been subject to some minimal editorial review or peer examination, guaranteeing at least a minimal level of quality. Code compliant with this criteria and still deemed illegal by a C∀ implementation cannot be dismissed as an example of poor C coding practices, but as evidence against the assumptions that informed the design of C∀.

The GNU project is a prime example of open-source code. Also, its repository is conveniently accessible and extensively mirrored. The code it hosts has no particular bias towards any application domain nor are particular restrictions placed on program size. However, this software collection is enormous. In 2002, Wheeler [136] estimated the size of a typical GNU/Linux system at over 30 million lines of code, 71% of which are written in C. Clearly some sampling is needed to keep these numbers within a manageable size for this experiment, and yet maintain a significant cross-section of application domains and program size, i.e., it is desirable for the study to include programs ranging from the size of the Unix word counter `wc` to that of database management systems.

The GNU project organizes its software in *packages*, each of which comprises code and documentation for a program or a related set of programs. The discussion that follows refers to packages as the unit of distribution. Packages that include a large number of executables are likelier to include a set of smaller complete programs than packages of similar size that include only one executable. It is reasonable to assume that complete programs are described in fewer source files, i.e., that shorter programs are contained in these packages than, say, `emacs`, a package containing thousands of source files that result in approximately ten executables. The information on the *executable density* of a package can also be obtained from the GNU directory.

The experimental sample comprises various application domains, ranging over such areas as language processing, communications, system tools, statistics, plotting, etc. Packages whose content is deemed too similar have been dropped from the examined set. An example of this are the `marst` Algol-to-C and the `cim` Simula-to-C translators, only the latter of which was included in the examined set.

Further criteria have been considered for the sampling process. In particular,

Package	Version	LOC	Description
Termutils	2.0	2812	Programs for controlling terminals
Wget	1.9	20688	A network utility to retrieve files from the Web
Gsl	1.4	146420	A collection of routines for numerical computing
Emacs	21.3	211129	The extensible display editor
Cim	3.30	22346	A compiler for the Simula language
Bison	1.875	19168	Parser generator (yacc replacement)
Patch	2.5.4	7147	Apply a diff file to an original source
Plotutils	2.4.1	71743	Utilities for plotting scientific data
Sed	4.0.9	18015	Stream Editor
Textutils	2.1	34662	Text utilities
Uucp	1.07	48341	File copying program
Total		602471	

Table 6.1: Selected packages

the *popularity* of a package has been used as a measure for tiebreaking. Popularity is a measure that reflects the use of a package in terms of the number of times it has been downloaded and the frequency of maintenance releases (it does not discriminate, however, between feature addition and error correction). Although not wholly reflective of the quality or even usage of the code (some widely used programs are famous for the irregularity of new releases), popularity is a good measure of how closely a package is examined and tested, and as such, became the deciding factor in the case of conflict between similar packages. For example, in the case of `cim` against `marst` described above, the more popular `cim` made it into the sample. Popularity information is maintained in the GNU repository itself (in the form of number of downloads) or, more systematically, by independent open-source projects directories, such as `freshmeat` [1].

The GNU Project lists 248 package in its directory [54]. 151 could be obtained automatically. Of these, 11 packages, comprising over 500,000 lines of code were selected. Lines of code (LOC) counts were obtained with the `c_count` [46] statistics gatherer, which parses C code and discards lines containing only comments in its results. The selected packages and their sizes are shown in table 6.1.

6.3 Code Analysis Infrastructure

For day-to-day programming, high-level code examination is often performed via generic text-searching tools, such as the well-known `grep`. However, the code patterns required by this analysis are considerably more complicated, and therefore more elaborate means are called for.

Tools for the analysis of source code, other than language translators, include tag generators, lines-of-code counters and a variety of others. In particular, tools to aid program understanding and reverse engineering are particularly well known. Cox's comparative survey [36] provides a comprehensive overview. The most noticeable difference between these tools is the representation they use for the analysed code. These representations range from relational databases (as in the case with AT&T Research Labs' `ciao` [83]) to structured text forms (like the XML specializations GXL [68], or CPPML). Cox adopted yet another approach in his Jupiter/Mercury system [35], which relies on the MultiText text-retrieval engine. MultiText provides efficient access to the gathered information, regardless of the size of the stored corpus, whereas Mercury supplies an expressive query language. The combination allows for fast and convenient searching of rich patterns in potentially vast quantities of code.

The particulars of the internal document representation, retrieval model and query language of MultiText are explained by Clarke, et al. [30]. It is enough for the purposes of the current discussion to say that MultiText tokenizes and annotates the document, and then indexes this extended stream of tokens. Annotations consist of *tags* that provide some information as to the semantic rôle of each token in the document. Jupiter relies on a compiler front-end (the combination lexer/parser/symbol table) to provide these tags.

In its original version, Jupiter includes a parser, `agc`, based on the Roskind grammar [110], which does not handle C99 or `gcc`-extensions. Since a good amount of the examined GNU code makes use of these features, the system could not be used "out-of-the-box". The modular architecture of Jupiter allows for the parser to be replaced by a more suitable one. Clearly, adapting the C∀ translator parser to perform this function would not be an independent measuring apparatus. A workable alternative is `ckit`, a C compiler front end from Bell Labs. Extensively tested and able to handle C99 and some `gcc` extensions, `ckit` has an extensible architecture that makes it possible to write plug-in modules, such as the code annotator for Jupiter, with relative ease[1].

The new annotator has a slightly richer set of tags than the original Mercury (it reintroduces several tags of the earlier Mars [37] system). In particular, besides decorating the source code, the new annotator keeps file and package information. This metainformation becomes relevant when discriminating certain aspects of the code, e.g., multiple inclusions of the same file.

Building a repository from a great number of sources is a tedious and error-prone task, so the highest degree of automation is desirable. First, a wrapper around the annotator was created that can be used as a plug-in replacement for the C compiler. This wrapper allows the reuse of the package-provided `Makefiles`.

Once the source code repository is built, the information extraction is done via the last component of the system, the Mercury interpreter. Mercury is a modified Scheme interpreter that subsumes the MultiText query language GCL. Queries expressed in GCL include "literal", e.g., the number of occurrences of a string within a file, as well as "structural" expressions, e.g., the number of uses of an identifier, as opposed to its declaration.

The process of building and using the repository is illustrated in figure 6.1. After packages have been obtained and configured (which is the only operation in the system that requires significant manual intervention[2]), they are built. Via the wrapped annotation generator, the build process transparently populates the MultiText backend database. This database stores the stream of tokens, and annotations and meta-annotations that are obtained from each source file after preprocessing.

[1]There is a very similar project, called `cil`. The ostensible difference with `ckit` is that `cil` is implemented in the competing dialect OCaml. The reasons `ckit` was chosen over `cil` for conducting the experiments described in this chapter are more circumstancial than technical. However, `ckit` has been more extensively tested with large bodies of code, whereas I was unable to determine whether the same was true for `cil`.

[2]Using the GNU Project for code samples presents the additional advantage that these programs conform to a common configuration and building mechanism, which suggests that this part of the process can be largely automated as well.

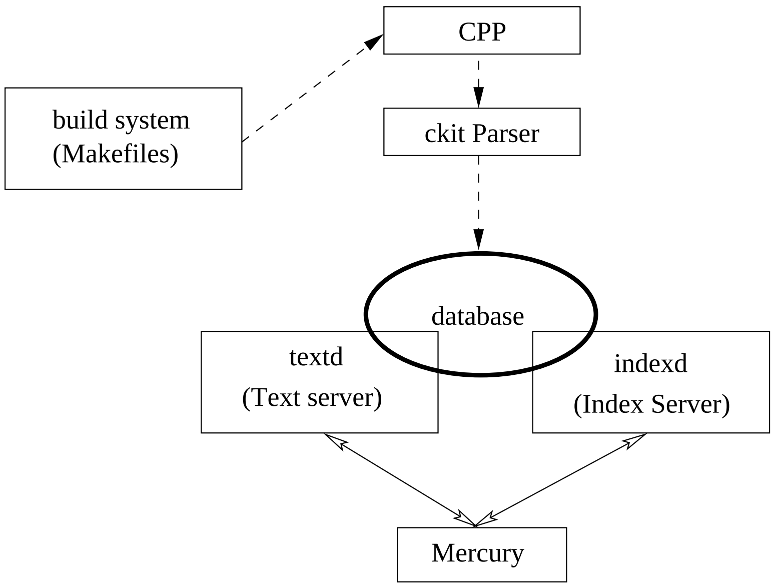

Figure 6.1: Source code analysis system

In the figure, the dashed lines represent the progress of a stream of data into the system. Once inside the repository, Mercury is used in batch mode to query the database and gather statistics (the solid lines in the figure represent query-answer dialogues).

6.3.1 Noise-introducing factors

A further problem in determining the exact number of compatibility errors is due to the fact that two or more files might textually include the same problematic C file. These errors include declaration of symbols that clash with new keywords. Errors of this kind would be reported as many times as a file is included. The metainformation included in the repository allows for a query to specify that each file is to be considered only once, thus eliminating this situation.

Automatically generated C programs can also slant results, as generators use the same form of output. Errors within these files are due more to the generator than to the use of C, so packages that involve the use of these kind of files should be handled especially. For the packages in table 6.1, the only code generators involved are `flex` and `bison`, whose output posed no particular problems.

6.4 Search patterns

With the infrastructure described above, the C code sample can be analysed with respect to the changes in C∀. As outlined in the introduction, there are two changes in C∀ that seem the most likely to cause incompatibility problems. The first is the the introduction of keywords that may be used as identifiers in valid C. The second are the modifications to the syntax of the **switch** statement, which in C may include any number of declarations and executable statements between the keyword **switch** and the first **case**, whereas C∀ only allows for the inclusion of declarations in this position. Also, C allows a **switch** to be interwoven with other control statements, a behaviour that C∀ disallows syntactically. Finally, understanding fall-through **case** clauses is interesting with respect to C∀ extensions. Uses of these constructs are searched for, to estimate their frequency in actual code.

Identifier clashes All the declarations of variables and functions are collected, and then an intersection of them with the 12 new C∀ keywords (**choose**, **context**, **dtype**, **fallthru**, **finally**, **forall**, **ftype**, **lvalue**, **resume**, **terminate**, **try** and **type**) is performed.

Switch frequency One pattern looks for all **switch** statements to determine how frequently they occur.

Unreachable code in switches statements between a **switch** and the first **case** (or **default**), either executable or declarations, are checked for.

Intertwined control statements Loop statements that are not completely contained within a **case** block are searched for.

Uses of falling-through cases **case**s that do not include a **break** or **return** before the next **case** in the **switch** are considered an instance of falling through **case**s.

Fall-through to simulate lists or ranges A **case** followed by another **case** without intervening statements are considered as an instance of **case** lists.

6.5 Results

Results are reported per package, because a package provides a better sense of how constructs are used in complete programs, rather than functions or files. Queries for the foreseen problematic patterns were submitted to the repository, with the following results:

Identifier clashes: In the 11 examined packages only the keywords **type** and **context** appeared as identifiers. The clashing identifiers are shown in the second column of table 6.2. For each clash, the first number denotes the number of declarations of the clashing identifier and the second number denotes the total number of identifiers declared in the package.

Switch frequency: **switch** frequency (compared against the total number of statements) is shown in the third column of table 6.2. The usage of **switch** statements is even lower than I had expected.

Unreachable code in switches: There was not a single occurrence of declarations or executable code after a **switch** but before the first corresponding **case**.

Intertwined control structures: In the code examined, there was not a single occurrence of a **switch** intertwined with another control structure (i.e., Duff's device).

Uses of falling-through cases: Occurrences of fall-through cases are reported in the fourth column of the table. The first number denotes the total number of consecutive **case** clauses *without* intervening statements (case lists), and the second number is the total number of occurrences of consecutive **case** clauses *with* intervening statements. Note that a little over half of the uses of fall-through **case**s are for **case** list/ranges.

Package	Identifier clashes	switch Frequency(%)	Fall through cases
Termutils		1.10395	36/47
Wget	type (2/66422)	0.52091	38/65
Gsl		0.01778	25/34
Emacs	type(16/115353)	0.63812	201/456
Cim	type (1/4262)	1.39616	58/129
Bison	context (3/16992)	0.59113	198/211
Patch	context (7/7948)	0.00608	30/58
Plotutils	type(22/5289)	0.70640	97/137
Sed		0.51948	23/35
Textutils	context (5/891)	0.64199	273/476
Uucp	type(3/3527)	0.51194	155/186

Table 6.2: Results

It is clear from the results that the most problematic issue when compiling ANSI C applications via the C∀ translator is the clashing of identifiers, in particular, the identifier **type**. Since the use of **type** is central to C∀'s parametric polymorphism machinery, and the use of any other word would lead to confusion, C∀'s designers decided to include the **type** keyword in the language, at the expense of minor adjustments in C legacy code. It is important to note that C++ introduced a similar number of new keywords and large systems of legacy C code were adjusted to work with it.

Next, the C∀ changes with respect to the **switch** statement do not present problems because programmers, in general are not using the features that were eliminated. It is my opinion that further examination of a larger sample complying with a set of coding standards would only confirm the results presented here in regards to declarations and code in between **switch**es and their first **case**, or the popularity of Duff's device and its variants. Their absence in the examined sample is indicative of their rarity in practical code.

Finally, the observed use of fall-through in **switch**es suggests that the combination of C∀'s **choose** and **case** lists and ranges cover half of the situations where this problematic feature is necessary, with benefits to readability and maintainability. Hence, these extensions appear to be warranted.

6.6 Related work

Source code analysis is normally associated with program understanding and reverse-engineering (for example, see [33]). However, there are some instances of empirical analysis of source-code corpora as a means to settle language design discussions, e.g., Wright's [139] study of ML source code resolved the dispute in regards to the introduction of the value restriction rule in the language. Particularly relevant to the study described in this chapter is Ernst et al. [52], which examined C preprocessor usage to determine the importance of preprocessor-aware tools for source-code analysis.

The existence of a large variety of dialects of the C language have motivated compatibility studies. The analysis of the compatibility between C and C++ has drawn, understandably, particular interest. Most of this studies, however, use a high-level approach, e.g. Stroustroup [124, 125] or Tribble [133]. To my knowledge, analysis of a large source-code base to measure compatibility with an extended language has not been done for C.

Chapter 7

Conclusions

William Bragg said that the important thing in Science is not so much to obtain new facts as to discover new ways of thinking about them. In a similar vein, much of this thesis is about selectively choosing a number of existing language ideas and blending (engineering) them together to produce a set of consistent, orthogonal and expressive extensions to C. Unfortunately, it is difficult to quantitatively measure the success of this work because all useful programming languages are Turing-complete and all Turing-complete programming languages are equivalent in some broad sense. Nevertheless, there are differences among programming languages because programmers often chose a languages to fit an application problem, implying one language has at least subjective advantages over another in certain specific situations. Given that many of the differences among programming languages are subjective, are there some general observations that can be made about what is good or bad?

There is strong evidence that supports the claim that the variety of problems computers are used to solve, and of programmers writing solutions for them, escape the scope of any single programming language. Different languages rely and implement a different "cluster of programming notions" [43]. Trying to make this cluster as large as possible for a particular language results in "kitchen sink" languages like PL/I and Ada, which are extremely powerful but have not enjoyed widespread acceptance.

An alternative to the “kitchen sink” approach for language design is to provide a set of powerful mechanisms and let programmers build libraries to generalize the language into multiple application domains. What is crucial in this approach is that the mechanism for extension must meld with the builtin language features and with the extensions generated by programmers, resulting in a seamless language system. C++ made an excellent attempt at this approach, integrating builtin and user-defined types within a complex, extensible type-system supporting inheritance, overloading and generics.

However, another component of C++’s success is its extension from the simple programming model of C, a language already popular, in consistent and unsurprising ways to make use of well-chosen abstractions. This school of design is the “evolutionary” approach, and it involves a more constrained design space than its “revolutionary” counterpart (like Java). Whereas the latter approach requires a clear notion of the new language and its abstraction, the former also presupposes a thorough understanding of the substrate language, both in design and how it is used. Careful syntactic extensions and a firm grasp of the interactions of the added abstractions are necessary to make the extensions fit with existing notions. Interestingly, the current growth of C++ reflects its adoption by more ambitious programming projects, and these projects are correspondingly using more of the advanced languages features (especially the generic Standard Template Library).

C∀ mimics the approach taken by C++ but adopts a different set of abstractions mechanisms to achieve its goals. First, C∀ attempts to remain backwards compatible with the large corpus of legacy C code (as did C++). Second, C∀ attempts to fix some of the more obvious flaws in C in an attempt to make the language accessible to a broader range of programmers, even beginning programmers (unlike C++). Third, C∀ introduces a number of new language abstractions and mechanisms that I believe significantly extend the power of the language and simplify the programming task.

C∀, as presented initially by Ditchfield [47], introduced to C a type system that provides for advanced overloading and polymorphic functions, while preserving the semantics of the existing type system. C∀ programmers can specify the behaviour of functions without extensionally refering to types but by intensionally describing types in terms of the operations the function uses. Bilson's implementation of this type system [16] introduces extensions (MVR functions) and artifacts that serve as starting points for the features described in this thesis.

In this thesis, both corrective and proactive changes have been made to C. In changing the **switch** statement I have revisited an existing control structure to prevent some egretious mistakes and misuses. In adding new syntax for declarations, new loop control statements and the **choose** statement, I have provided more structured and less error-prone mechanisms that can co-exist with their traditional counterparts for the sake of legacy code. In adding tuples, exceptions and attributes, I have enabled new ways of writing programs that can reduce the effort involved and increase robustness. Finally, I have demonstrated that these changes have minimal impact on existing C programs.

The litmus test of the effectiveness of a programming language is usage. Currently there is only a very small amount of C∀ code, in the form of test programs used to ascertain the correctness of specific mechanisms. There are no medium/large programs to demonstrate the applicability of these mechanisms to specific programming tasks. Nevertheless, languages with similar facilities, like Java, C++ and ML, have shown that similar features are useful and usable, but only time and use will tell the true effectiveness of these features, as well as the tasks they are most adequate for.

7.1 Future Work

Although C∀ goes a long way toward fixing the most insidious shortcomings of C, there are still areas that need addressing. First, the built-in support for arrays in C is error-prone and difficult for beginners to understand, but is so deeply ingrained in the language that it is impossible to change in any significant way. A better *ad hoc* array mechanism is feasible, but I feel a more general solution, *generic types*, is preferable. Such an addition to the C∀ type system would permit the encapsulation of adequate array handling facilities in a library, as well as provide support for generic container libraries. To this end, some initial design work has been done on generic types for C∀, but it is still far from completion.

Two other glaring defects in C are currently unaddressed by C∀: concurrency mechanisms, and advanced support for modularity. Concurrency opens the door for a new way of thinking about programming and structuring programs. Modularity facilitates the development of large systems. Of the two, the one that requires the greatest design effort is, without a doubt, concurrency, given the diversity of concurrency mechanisms and approaches. Both concurrency and modularity are crucial for modern programming tasks, and increase the possibilities of success for a language.

Also crucial for the success of a language is the availability of development tools. Features like attributes, that allow for additional communication between the program and language processing systems, allow for powerful tools. Traditional support tools like debuggers and profilers are likely to be similar to those in existence, but the sophisticated type system incorporated in C∀ will likely require a constructive tool that furnishes programmers with information about the overloading resolution and type specialization process. To this end, I am developing an explainer, in the spirit of the one described by Duggan [50], that presents the process of type selection in a meaningful way.

Another area that remains unexplored in this work is the optimization that the new forms in the language allow. The expressiveness gained in C∀ indicates that there is more information available to an optimizer for generating more efficient code. To exploit these hypothetical gains, however, would require the creation of a true C∀ compiler, rather than the current translator approach. In addition, the sound theoretical basis that the C∀ type system provides is likely to help verify the correctness of code optimizations.

Finally, the C∀ type system is built upon an elegant formalism. Wanting equally solid theoretical standing for the extensions presented in this thesis, I looked for appropriate formalisms to cover tuples, exceptions and attributes. Interestingly, I believe now that I have found a single formalism that covers all these features. The intuitionistic modal logic S4, when mapped into a type system, encompasses the static and dynamic aspects of all the extensions discussed in the text. Experimental languages like Template Haskell and MetaML, based on variants of this type system are becoming increasingly important within the research community, and are also starting to draw the attention of practitioners, since they provide a solid foundation for persistence, distributed programming and other dynamic programming tasks. If the C$\forall$ type system is extended in this direction, not only would the benefits of a more dynamic style of programming be reaped, but they would also be attained without detriment to legacy code, or a steep learning curve. A language traditionally considered low- to mid-level, that could traverse the whole spectrum of computation and stand together with languages that enable the highest level of abstraction, would be the ultimate argument for the evolutionary approach of programming language design.

Bibliography

[1] freshmeat Unix and cross-platform Software Directory. `http://freshmeat.net`.

[2] The packet design embedded library. `http://www.dellroad.org/pdel`.

[3] Risks the programming language is accountable for. `http://catless.ncl.ac.uk/Risks/9.69.html`.

[4] From C to C++: Interviews with Dennis Ritchie and Bjarne Stroustrup. *Dr. Dobb's Journal C Sourcebook*, 1990.

[5] The `libunwind` project. `http://www.hpl.hp.com/research/linux/libunwind/`, December 2003.

[6] ISO/IEC 9899. *Programming languages – C*. 1999.

[7] Andrei Alexandrescu. *Modern C++ design*. Addison-Wesley Publishing Company, 2001.

[8] Eric Allman and David Been. An exception handler for C. In *USENIX Association Summer Conference Proceedings*, pages 25–45, Portland. Oregon, USA, June 1985.

[9] American National Standards Institute. Information Processing Systems Committee X3 and Computer and Business Equipment Manufacturers Association. Rationale for draft proposed American National Standard for information systems: programming language C: X3J11/88-15: Project: 381-D. Technical report, 1988.

[10] Bruce Anderson. Type syntax in the language C: An object lesson in syntactic innovation. *ACM SIGPLAN Notices*, 15(3):21–27, March 1980.

[11] Andrew W. Appel and David B. MacQueen. Separate compilation for standard ML. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, 1994.

[12] J. Michael Ashley and R. Kent Dybvig. An efficient implementation of multiple return values in Scheme. In *1994 ACM Conference on LISP and Functional Programming*, June 1994.

[13] Lennart Augustsson. Cayenne, a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, 1998.

[14] Matt H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley Publishing Company, 1999.

[15] D.W. Barron, J.N. Buxton, D.F. Hartley, E. Nixon, and C. Strachey. The main features of CPL. *The Computer Journal*, 6(2):134–143, July 1963.

[16] Richard Bilson. Implementing overloading and polymorphism in Cforall. Master's thesis, University of Waterloo, Waterloo, Ontario, February 2003.

[17] Joshua Bloch. JSR 175: A metadata facility for the Java programming language. `http://www.jcp.org/en/jsr/detail?id=175`.

[18] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In Nick Benton and Andrew Kennedy, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[19] The boost library. `http://www.boost.org`, 2000.

[20] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[21] Timothy A. Budd. An implementation of generators in C. *Computer Languages*, 7(2):69–87, 1982.

[22] Peter Buhr, Ashif Harji, and W.Y.Russel Mok. Advanced exception handling. *IEEE Transactions on Software Engineering*, 26(9):820–836, September 2000.

[23] Peter A. Buhr. A case for teaching multi-exit loops to beginning programmers. *SIGPLAN Notices*, 20(11):14–22, Nov 1985.

[24] Peter A. Buhr. Are safe concurrency libraries possible? *Communications of the ACM*, 38(2):117–120, February 1995.

[25] Peter A. Buhr, Hamish I. Macdonald, and C. Robert Zarnke. Synchronous and asynchronous handling of abnormal events in the μSystem. *Software—Practice and Experience*, 22(9):735–776, 1992.

[26] Peter A. Buhr, David Till, and C.R. Zarnke. Assignment as the sole means of updating objects. *Software—Practice and Experience*, 24(9):835–870, September 1994.

[27] D. Cameron, P. Faust, D. Lenkov, and M. Mehta. A portable implementation of C++ exception handling. In *Proceedings of the C++ Conference*, pages 225–243. USENIX Association, August 1992.

[28] Morten Mikael Christensen. Methods for handling exceptions in object-oriented programming languages. Master's thesis, Odense University, Denmark, January 1995.

[29] Tyng-Ruey Chuang, Chuan-Chieh Jung, Wen-Min Kuan, and Y. S. Kuo. Objectstream: Generating stream-based object I/O for C++. In *24th International Conference on Technology of Object-Oriented Languages and Systems*, pages 81–90, Beijing, China, September 1997.

[30] Charles L.A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

[31] The C Language Standardization Committee. The C language standardization charter. `http://std.dkuug.dk/JTC1/SC22/WG14/www/charter`.

[32] Curtis R. Cook. Information theory metric for assembly language. *Software Engineering Strategies*, pages 52–60, March/April 1993.

[33] Tama Communication Corporation. Source code tours. `http://www.tamacom.com/tour.html`.

[34] Adam M. Costello and Cosmin Truta. CEXCEPT exception handling in C. `http://cexcept.sourceforge.net`.

[35] Anthony Cox. Jupiter user manual and Mercury language reference. Draft.

[36] Anthony Cox and Charles Clarke. A comparative evaluation of techniques for syntactic level source-code analysis. In *7th Asia-Pacific Software Engineering Conference (APSEC)*, Singapore, December 2000.

[37] Anthony Cox and Charlie Clarke. A functional approach to complex retrieval tasks. In *Ninth International Workshop on Functional and Logic Programming*, pages 486–498, Benicassim, Spain, September 2000.

[38] Brad J. Cox and Andrew J. Novobilski. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley Publishing Company, Reading, Mass., USA, 1991.

[39] C.T.Zahn. *C Notes: A guide to the C programming language*. Yourdon Press, 1979.

[40] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. Addison-Wesley Publishing Company, 2000.

[41] Laurent Dami. *Object-Oriented Software Composition*, chapter Functions, Records and Compatibility in the λ N-Calculus, pages 153–174. Prentice Hall, 1995.

[42] C. J. Date. *An introduction to database systems: vol. I (4th ed.)*. Addison-Wesley Publishing Company, 1986.

[43] Jaco de Bakker and Erik de Vink. *Control flow semantics.* MIT Press, 1996.

[44] Universidad de Oviedo's Computational Reflection Group. The nitrO system. `http://www.di.uniovi.es/reflection/lab`.

[45] Robert B.K. Dewar and Edmond Schonberg. The elements of SETL style. In *Proceedings of the 1979 annual conference*, pages 24–32. ACM Press, 1979.

[46] Thomas E. Dickey. `c_count`, the C language line counter. `http://dickey.his.com/c_count/c_count.html`.

[47] Glen Jeffrey Ditchfield. *Contextual Polymorphism.* PhD thesis, University of Waterloo, 1994.

[48] Véronique Donzeau-Gouge, Gilles Kahn, and Bernard Lang. Formal definition of the Ada programming language. Technical report, Honeywell, CII Honeywell Bull, INRIA, November 1980. (preliminary).

[49] Rémi Douence and Mario Südholt. On the lightweight and selective introduction of reflective capabilities in applications. In *ECOOP'00 Workshop on Reflection and Meta-Level Architectures*, 2000.

[50] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

[51] Daniel Edelson and Ira Pohl. C++: Solving C's shortcomings? *Computer Languages*, 14(3):137–152, September 1989.

[52] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, December 2002.

[53] Michael J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6(3/4):2599–288, November 1993.

[54] Free Software Foundation. FSF/UNESCO Free Software Directory. `http://www.gnu.org/directory/`.

[55] Daniel P. Friedman and David S. Wise. Functional combination. *Computer Languages*, 3(1):31–35, 1978.

[56] Richard P. Gabriel. Worse is better. `http://www.dreamsongs.com/WorseIsBetter.html`.

[57] Richard P. Gabriel. The end of history and the last programming language. *Journal of Object Oriented Programming*, 20(7), July 2002.

[58] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 115–134, 2003.

[59] Jacques Garrigue and Hassan Aït Kaci. The typed polymorphic label-selective λ-calculus. In *Proc. of the 21st ACM Symposium on Principles of Programming Languages*, Portland, 1994.

[60] John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12), December 1975.

[61] Paul Graham. *ANSI Common Lisp.* Prentice Hall, 1995.

[62] Mark Grossman. Object I/O and runtime type information via automatic code generation in C++. *Journal of Object Oriented Programming*, 6(4):34–42, July/August 1993.

[63] The OMG Group. C language mapping specification. `http://www.omg.org/cgi-bin/doc?formal/99-07-35`, 1999. (OMG formal/99-07-35).

[64] W.T. Hardgrave. Positional versus keyword parameter communication in programming languages. *ACM SIGPLAN Notices*, 11(5):52–58, May 1976.

[65] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. In *Conf. Record 18th Ann. ACM SIGPLAN-SIGACT Symp. on POPL 91*, Orlando, FL, January 1991.

[66] C.A.R. Hoare. Hints on programming language design. *ACM Symposium on Principles of Programming Languages*, pages 1–30, 1973.

[67] C.A.R. Hoare. *ACM Turing Award Lectures. The first twenty years 1966–1985*, chapter The emperor's old clothes. ACM Press/Addison-Wesley Publishing Co., 1987.

[68] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Toward a standard exchange format. In *Seventh Working Conference on Reverse Engineering*, pages 162–171, Brisbane, Queensland, Australia, November 2000.

[69] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*, 2001.

[70] Samuel Harbison III and Guy Lewis Steele Jr. *C A reference manual.* Prentice Hall, 5th edition, 2002.

[71] International Organization For Standardization, International Electrotechnical Comission, Intermetrics Inc. *Annotated Ada Reference Manual*, v6.0 edition, December 1994. ISO/IEC 8652:1995.

[72] Jaakko Järvi. Tuple types and multiple return values. *C/C++ Users' Journal*, 19:24–35, August 2001.

[73] Jaakko Järvi and Bjarne Stroustrup. Mechanisms for querying types of expressions: `decltype` and `auto` revisited. Technical Report N1571, Proposal to the C++ Standard Committee.

[74] Xing Jin. Intensional C compiler. `http://i.csc.uvic.ca/~jinxing/icc`.

[75] Mark P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, 2000.

[76] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[77] Brian W. Kernighan. A descent into Limbo. `http://www.vitanuova.com/inferno/papers/descent.html`.

[78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, first edition, 1978.

[79] Gregor Kiczales, Jim Des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[80] Jorgen Lindskov Knudsen. *Advances in Exception Handling Techniques*, volume 2022 of *Lecture Notes in Computer Science*, chapter Fault Tolerance and Exception Handling in BETA, pages 1–17. Springer-Verlag, 2000.

[81] Donald E. Knuth. Structured programming with go to statements. *ACM Computing Surveys*, 6(4):261–301, December 1974.

[82] Andrew Koenig. C traps and pitfalls. Technical report, Bell Labs, July 1986. CSTR #123.

[83] Balachander Krishnamurthy, editor. *Practical reusable UNIX software*. John Wiley & Sons, Inc., 1995.

[84] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 1962.

[85] K. Läufer and M. Odersky. Self-interpretation and reflection in a statically typed language. In *Proc. OOPSLA Workshop on Reflection and Metalevel Architectures*. ACM, October 1993.

[86] Doug Lea. *User's Guide to the GNU C++ Library (version 2.0)*, April 1992.

[87] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., 1993.

[88] Barbara Liskov. A history of CLU. *ACM SIGPLAN Notices*, 28(3):133–147, 1993.

[89] John Maddock and Steve Cleary. C++ type traits. *Dr. Dobb's Journal*, October 2000.

[90] Jan Messerschmidt and Reinhard Wilhelm. Constructors for composed objects. *Computer Languages*, 7(2):53–59, 1982.

[91] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual (version 5.0). Tech. Report CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, CA, USA, April 1979.

[92] R. P. Mody. C in education and software engineering. *ACM SIGCSE Bulletin*, 23(3), September 1991.

[93] P.J. Moylan. The case against C. Technical Report EE9240, Centre of Industrial Control Science, Department of Electrical and Computer Engineering, University of Newcastle, Australia, July 1992.

[94] Frank Mueller and David B. Whalley. Avoiding unconditional jumps by code replication. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 322–330. ACM Press, 1992.

[95] Nathan C. Myers. Traits: a new and useful template technique. `http://www.cantrip.org/traits.html`.

[96] Matthias Neubauer and Michael Sperber. Down with Emacs Lisp: dynamic scope analysis. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 38–49, 2001.

[97] W. W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat and exit statements. *Communications of the ACM*, 16(8):503–512, Aug 1973.

[98] Matt Pietrek. A crash course on the depths of Win32 structured exception handling. *Microsoft Systems Journal*, January 1997.

[99] Rob Pike. How to use the Plan 9 C compiler. Technical report, Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ, USA, 2000.

[100] Gordon Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[101] Ira Pohl and Daniel R. Edelson. A–Z: C language shortcomings. *Computer Languages*, 13(2), 1988.

[102] Norman Ramsey and Simon Peyton-Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 285–298. ACM Press, 2000.

[103] Jean-Claude Raoult and Ravi Sethi. Properties of a notation for combining functions. *Journal of the Association for Computing Machinery*, 30(3):595–611, 1983.

[104] Derek Rayside and Gerard T. Campbell. An Aristotelian understanding of Object-Oriented programming. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 337–353. ACM Press, 2000.

[105] Martin Richards and Colin Whitby-Stevens. *BCPL – The language and its compiler*. Cambridge University Press, 1979.

[106] Dennis M. Ritchie. The development of the C programming language. In Thomas G. Bergin and Richard G. Gibson, editors, *History of Programming Languages*. ACM, Addison-Wesley Publishing Company.

[107] Dennis M. Ritchie. Very early C compilers and language. `http://www.cs.bell-labs.com/who/dmr/primevalC.html`.

[108] Dennis M. Ritchie. The development of the C language. In *The second ACM SIGPLAN conference on History of programming languages*, pages 201–208. ACM Press, 1993. `http://cm.bell-labs.com/cm/cs/who/dmr/chist.html`.

[109] Eugene J. Rollins. Sourcegroup: a selective-recompilation system. In Robert Harper, editor, *Third International Workshop on Standard ML*, September 1991.

[110] Jim Roskind. LALR(1) C Grammar. `http://www.empathy.com/pccts/roskind.html`.

[111] Tom Schotland and Peter Petersen. Exception handling in C without C++. *Dr. Dobb's Journal*, pages 102–112, November 2000.

[112] Ravi Sethi. Uniform syntax for type expressions and declarators. *Software—Practice and Experience*, 11(6):623–628, June 1981.

[113] Andrew Shalit. *The Dylan Reference Manual.* Apple Press/Addison Wesley, 1996.

[114] Mary Shaw, William A. Wulf, and Ralph L. London. Abstraction and verification in alphard: defining and specifying iteration and generators. *Communications of the ACM*, 20(8):553–564, 1977.

[115] Tim Sheard. Automatic generation and use of abstract structure operators. *ACM Trans. Program. Lang. Syst.*, 13(4):531–557, 1991.

[116] Dorai Sitaram. unwind-protect in portable Scheme. In Matthew Flatt, editor, *Proceedings for the 4th Workshop on Scheme and Functional Programming*, pages 48–52, University of Utah, November 2003. `http://www.ccs.neu.edu/home/dorai/uwcallcc/uwcallcc.html`.

[117] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM, January 1984.

[118] Eric S.Raymond, Guy L. Steele Jr., and Richard Stallman et al. The Jargon file. `http://www.catb.org/~esr/jargon/`.

[119] Guy Steele. Growing a language. In *OOPSLA '98*, October 1998. Invited talk.

[120] D. Stemple, R. Morrison, G.N.C. Kirby, and R.C.H. Connor. Integrating reflection, strong typing and static checking. In *16th Australian Computer Science Conference (ACSC'93), Brisbane, Australia*, pages 83–92, 1993.

[121] Bjarne Stroustrup. XTI: The extended type information library. Technical report, AT&T Labs – Research.

[122] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Publishing Company, 1994.

[123] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 3 edition, 1997.

[124] Bjarne Stroustrup. C and C++: Case studies in compatibility. *C/C++ Users' Journal*, 20(9):22–31, September 2002.

[125] Bjarne Stroustrup. C and C++: Siblings. *C/C++ Users' Journal*, 20(7), July 2002.

[126] Stefano Taschini, Markus Emmenegger, Henry Baltes, and Jan G. Korvink. Smart enumeration in C++: virtual construction, message dispatching and tables. *Software—Practice and Experience*, 29(1):67–76, 1999.

[127] The Cyclone Programming Language Team. Cyclone homepage. `http://www.cs.cornell.edu/projects/cyclone`.

[128] The GTK+ team. GObject reference manual (for glib 2.5). `http://developer.gnome.org/doc/API/2.0/gobject/`, March 2004.

[129] Robert D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:97–112, 1977.

[130] Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-order and Symbolic Computation*, 15(2/3):141–160, 2002.

[131] David Till. Tuples in imperative programming languages. Master's thesis, University of Waterloo, 1989.

[132] Andrew Peter Tolmach. *Debugging standard ML*. PhD thesis, 1992.

[133] David R. Tribble. Incompatibilities between ISO C and ISO C++. `http://david.tribble.com/text/cdiffs.htm`, August 2001.

[134] Daveed Vandevoorde. Reflective metaprogramming in C++. Technical Report N1471, Proposal to the C++ Standard Committee, April 2003.

[135] Ben Werther and Damian Conway. A modest proposal: C++ resyntaxed. *ACM SIGPLAN Notices*, 31(11):74–82, 1996.

[136] David A. Wheeler. More than a gigabuck: Estimating GNU/Linux's size. `http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`, July 2002.

[137] Edward D. Willink and Vyacheslav B. Muchnick. Preprocessing C++: Substitution and composition. `citeseer.ist.psu.edu/259759.html`.

[138] Gregory V. Wilson. Extensible programming for the 21st century. `http://www.third-bit.com/~gvwilson/xmlprog.html`.

[139] Andrew K. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4):343–355, 1995.

[140] W.A. Wulf. BLISS: A language for systems programming. *Communications of the ACM*, 14(12):780–790, December 1971.

[141] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM Press, 1999.

Appendix A

Miscelaneous facilities

C∀ contains a number of minor features that ease the task of programming, or facilitate the reading and maintenability of C∀ programs. These are extended numeric literals and compound literals for the initialization of complex structures.

A.1 Numeric Literals

In the interest of ease of readability and inspired in a similar feature found in Ada, underscores embedded within a numerical constant are allowed. As most cultures have a similar construct, usually in the form of a comma or a period, this addition makes reading and typing long constants easier. This form of literals is invalid in C, so the addition is backwards compatible. All numeric literal forms in C99 are allowed in C∀. Examples:

```
2_147_483_648 // decimal constant
56_ ul // decimal unsigned long constant
0_377 // octal constant
0_ x_ ff_ ff; // hexadecimal constant
0x_ ef3d_aa5c / * hexadecimal constant * /
3.141_ 592_654 / * floating point constant * /
10_e_+1_00 / * floating point constant * /
0x_ ff.ff; // hexadecimal floating point
0x_ 1.ffff_ ffff_ p_128_ l // hexadecimal floating point long constant
```

Sequence of underscores, or underscores at the beginning or end of a sequence of digits are not permitted in numeric constants. Notice that _2, or _2_, for example, are valid identifier names. A numeric prefix may end with an underscore; a numeric infix may begin and/or end with an underscore; a numeric suffix may begin with an underscore. For example, the octal 0 or hexadecimal 0x prefix may end with an underscore: 0_377 or 0x_ff; the exponent infix E may start or end with an underscore 1.0_E10, 1.0E_10 or 1.0_E_10. Type suffixes U, L, etc., may start with an underscore: 1_U, 1_ll or 1.0E10_f.

A.2 Initializers

The C language did not use to have denotations for most *aggregates* (unions, arrays and structures), a situation that drove programmers to assign values to their members in a one-by-one fashion, a practice that is tedious, error-prone and that can obscure the intent of the program, by interleaving initialization code with the rest of the algorithm. These concerns become specially important when sparse aggregates are used in a program, which is often the case, but they do with high frequency in several problem domains, for example, numerical analysis.

In view of this situation, the C99 committee decided to incorporate some form of compound literals in the data initialization sublanguage, probably inspired by the of a *designated initializers* extension included in the gcc compiler. Although these additions can be perceived as mere syntactic sugar, compound denotations can expedite compilation and even run-time efficiency, since (with a proper implementation) aggregate assignments can be done with little or no analysis, whereas the alternative element-wise assignment, almost certainly done with nested loops, can only be optimized by some non-trivial data-flow analysis [90].

C89 initializers consist of lists of scalar literals or compile-time constant expressions (another C99 extension allows these expressions to be non-constant). The idea behind these lists is to represent visually the approximate layout of the object in memory. In C99, each element of the initializer list can be optionally *designated*, that is, identified by either a name or a number. This identifier must correspond to the names of one of the aggregate members of the aggregate being initialized, or, in the case of arrays, a number (a single number, since no "slicing" is allowed) within its boundaries.

Designated and non-designated initializers can be freely intermixed. The exact result of each initializer specification depends on the kind of entity being initialized. In the case of an **struct**, a positional initializer following a designated one will correspond to the position immediately after that of the designated initializer. C∀ allows for designated initializers, and extends the designator facility to include *tuple* designations.