# THE UNICONC UNICON COMPILER

BY

MICHAEL D. WILDER

A thesis submitted to the Graduate School

in partial fulfillment of the requirements

for the degree

Master of Science

Subject: Computer Science

New Mexico State University

Las Cruces, New Mexico

May 2006

“The Uniconc Unicon Compiler,” a thesis prepared by Michael D. Wilder in partial fulfillment of the requirements for the degree, Master of Science, has been approved and accepted by the following:

Linda Lacey
Dean of the Graduate School

Clinton L. Jeffery
Chair of the Examining Committee

Date

Committee in charge:

Dr. Clinton L. Jeffery, Chair

Dr. Gary A. Eiceman

Dr. Joseph J. Pfeiffer, Jr.

DEDICATION

## ACKNOWLEDGMENTS

This research was supported in part by an appointment to the National Library of Medicine Research Participation Program. This program is administered by the Oak Ridge Institute for Science and Education for the National Library of Medicine.

# VITA

## Professional Societies

## Publications

## Field of Study

Major field: Computer Science

ABSTRACT

THE UNICONC UNICON COMPILER

BY

MICHAEL D. WILDER

Master of Science in Computer Science

New Mexico State University

Las Cruces, New Mexico, 2006 v01b

Dr. Clinton L. Jeffery, Chair

Goal-directed programming languages present unique challenges in many areas for compiler developers. Substantial research in recent years has focused on the development of compilers for object-oriented languages, but little research has been published regarding compiler development for object-oriented, goal-directed languages. This thesis describes the design and development of a compiler for the object-oriented, goal-directed programming language Unicon.

TABLE OF CONTENTS

# LIST OF TABLES

LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

## 1.1 Background

### 1.1.1 The Icon Programming Language

The Icon programming language [Gris-79] is a very high-level, goal-directed, procedural programming language derived from the SNOBOL4 [Gris-71] and SL5 languages. Icon provides a rich set of high-level built-in data types that permit the rapid development of applications with a minimum of programming effort. Variables in Icon are untyped, and need not be declared before their use. Icon programs are usually reduced to bytecode by Icont, the Icon translator, and interpreted by Iconx, the Icon executive. Work on the Icon language proper was halted with the retirement of its creator circa 1994.

#### 1.1.1.1 Language Considerations

Icon programs are composed of declarations and expressions. The result of evaluating an Icon expression is effectively a tuple consisting of a value and a signal [Gris-79]. The value resulting from the evaluation of an Icon expression is used in the same manner as values in traditional programming languages. The signal resulting from the evaluation of an Icon expression is used to direct the flow of control in an Icon program. The unique nature of expression results in Icon is what distinguishes the language from its progenitors and its peers. The resulting semantics of Icon's expression evaluation is that an Icon expression can result in a single value (expression failure), no value, or a sequence of values.

Facilities for data abstraction in Icon are minimal. Abstraction in Icon is accomplished through the use of procedures and records. The emphasis on expressive control over data abstraction by the designers of Icon resulted in a language that is curiously powerful and relatively easy to use. Icon programs are constructed rapidly and are usually prototypical or experimental in nature. The uniquely expressive nature of Icon programs poses multiple challenges when transforming Icon expressions into statements or expressions in traditional programming languages.

#### 1.1.1.2 The Icon Interpreter

The virtual machine (VM) used to interpret Icon programs is as unique as Icon itself. The goal-directed nature of Icon typically requires a VM capable of backtracking as a result of expression failure. Backtracking in the Icon VM is facilitated by a stack of expression frames.

The instruction set of the Icon VM is relatively small, but extremely powerful and highly orthogonal. Variables in Icon are untyped and need not be declared before being referenced. The Icon interpreter must check the types of all instruction operands at the time of instruction execution in order to determine

```
struct descrip { /* descriptor */
    word dword; /* type field */
    union {
        word integr; /* integer value */
        char *sptr; /* pointer to character string */
        union block *bptr; /* pointer to a block */
        dptr descptr; /* pointer to a descriptor */
    } vword;
};
```

Figure 1.1: Descriptor Declaration

the feasibility of a requested operation. Instruction operands are presented to the Icon VM as descriptors that generically denote the type and value of a given operand. The declaration of a descriptor is shown in Figure 1.1. Many instructions in the Icon VM are manifested as functions in the run-time library of a given Icon implementation. The use of descriptors as operands to Icon VM instructions results in run-time library functions that contain multiple branches to resolve the destination of control flow depending upon the type or types of operands to a given VM instruction.

### 1.1.2 The Iconc Optimizing Compiler

The Iconc optimizing compiler [Walk-91] was the result of an extensive effort to support optimization experiments on Icon and other programming languages. Iconc consists of a run-time library and the compiler proper. The run-time library used by Iconc is implemented using a special-purpose language based upon the C programming language. This language, called the Iconc run-time library *implementation language* contains grammar constructs used to specify the types received, created, and returned by Icon built-in functions and operators. The information provided by these constructs is used during type inferencing. Iconc translates programs expressed in the Icon programming language into the C programming language. The C code generated by Iconc is compiled by a host C compiler, and is linked to the Icon run-time library by a host linker to create an executable program for the host platform. Iconc infers at compile-time the set of run-time types that an Icon expression may assume. The Iconc type inferencer is described in Section 1.1.2.2. The type inferencing model employed by Iconc precludes the separate compilation of Icon programs.

#### 1.1.2.1 Syntax Analysis

Syntax analysis in Iconc is performed by an LALR parser generated by YACC. Iconc constructs a parse tree and decorates it with attributes pertaining to a given node. Symbol tables for procedures, records, globals, locals, fields, and constants encountered in a given program are populated at this time. Checks for undefined symbol references and inconsistent redeclarations are performed by Iconc during this phase. The nodes of the tree that Iconc creates during syntax analysis contain member fields that are specifically designed for use during semantic analysis.

#### 1.1.2.2 Semantic Analysis

Iconc employs a type inferencing system based upon global dataflow analysis to determine the types that variables may assume during program execution. The parse tree that Iconc creates during syntax analysis is used to maintain a graph depicting the flow of data in an Icon program during type inferencing. Type information in Iconc is represented as a vector of bits. There are three separate sizes of bit vectors that are used to represent types at given program points depending on the set of types that can be generated at or propagated along a given program point. The size of bit vectors is determined at the start of type inferencing based upon information gathered during syntax analysis.

Before type inferencing proper begins, Iconc iterates over each procedure defined in a given program searching for code points where structures may be created during the execution of the program. At each such point, Iconc allocates a datum recording the characteristics of the structure that may be created and stores this datum in the parse tree node associated with the program point where creation may occur. Iconc again iterates over each defined procedure in order to allocate stores representing changes to be propagated along the flow graph during type inferencing. Each node of the flow graph representing a procedure is decorated with a store capable of representing the types that are propagated into the procedure, and a store representing the types that are propagated out of the procedure. If a procedure is capable of suspending, it is also decorated with a separate store capable of representing the types that the procedure generates as a result of being suspended. Stores are represented as arrays of pointers to type vectors. When each procedure in a program is visited during this pass, Iconc traverses the parse tree pertaining to the body of the procedure in order to determine the location of implicit and explicit loops that may survive successive iterations of type inferencing. For each such loop that is encountered, Iconc allocates a separate store capable of representing the types that will be propagated along the flow graph of the loop during type inferencing.

Type inferencing is performed in a sequence of iterations over a given Icon program. An iteration begins at the program entry point `procedure main()` and proceeds along the flow of execution until this procedure is exited. A running count of the number of changes propagated along flow graph edges is maintained during each type inferencing iteration. Type inferencing continues until the number of changes propagated during an iteration is zero. Each possible path of execution in a program is followed in an iteration of type inferencing. The computation performed during an iteration of type inferencing is exacerbated by the goal-directed nature of Icon semantics. The reader is directed to [O'Ba] and [Walk-91] for discussions of the intricacies posed by goal-directed evaluation in determining the lifetime of temporary variables in Icon programs.

#### 1.1.2.3 Optimizations

Iconc is capable of performing multiple optimizations on the internal representation of the C code that it generates for a given Icon program. The optimizations that Iconc can perform fall into two broad categories: control flow optimizations and invocation optimizations. Control flow optimizations are realized by performing peephole analysis on the intermediate C code after type inferencing of a program is completed. The peephole analyzer in Iconc eliminates certain forms of unreachable code, collapses branch chains, and removes unreferenced labels.

Iconc can also optimize the invocation of procedures and functions. These optimizations are particularly daedal. The interaction between these is a source of their complexity.

Invocation optimizations are performed by Iconc based upon information gathered from type inferencing, and information gathered regarding the lifetime of temporaries used as invocation parameters. Iconc is capable of inlining functions contained in the run-time library. If the developer of a run-time function indicates that inlining should be enabled, Iconc will inline code for the operation if and only if the target of the inlining contains no more than one type check. Some control flow optimizations performed by Iconc may result in the elimination or simplification of the signals emitted by an Icon expression. The reduction or patching of signal-handling code resulting from these optimizations is perfomed immediately before generated code is written.

#### 1.1.2.4 Code Generation

The Iconc code generator emits code to a single .c and its associated .h file. Code generation begins by writing C code corresponding to globals, statics, declarations, and literal constants in a given Icon program. The list of procedures contained in a given program is traversed, and code is generated for the declaration and body of each procedure if the procedure is within the potential path of control flow of the program. Information regarding the reachability of procedures is collected and recorded during type inferencing.

### 1.1.3 The Unicon Programming Language

The Unicon programming language [Jeff] is a very high-level, object-oriented, goal-directed programming language that is a descendant of and superset of the Icon. Unicon evolved over a number of years from a line-oriented preprocessor called "Idol" (Icon-Derived Object Language) into its present form as a full-fledged translator. Unicon preserves the syntax, semantics, and spirit of Icon while unobtrusively augmenting the data abstraction capabilities of Icon. Unicon is well suited for developing "one-shot" experimental programs because it honors its ancestry, and is equally well suited for the construction of large-scale software systems because it adds object-oriented grammatical constructs that readily facilitate encapsulation and code reuse.

Unicon encourages the use of namespaces in varied forms, thereby clearing the global namespace and potentially reducing coupling. Two such namespaces provided by Unicon are `class` and `package`. These namespaces require additional analysis in order to properly resolve the scope of a given symbol and to detect potential namespace collisions. The requisite additional analysis associated with namespaces is performed by Unicon during the syntax and semantic analysis phases.

Unicon translates Unicon programs into a dialect of Icon. The dialect of Icon that serves as the output of the Unicon translator is an extension of Icon proper. The Icon translator (Icont), executive (Iconx), and run-time library have been modified to accomodate the extensions that have been made since the retirement of Icon proper circa 1994. Unicon programs are interpreted using the updated versions of Icont, Iconx, and the run-time library.

#### 1.1.3.1 Syntax Analysis

Syntax analysis in Unicon is performed by an LALR parser generated by a modified version of the Berkeley YACC. The Unicon grammar is similar to the Icon grammar, but the areas where Unicon diverges are significant. The parse tree produced by Unicon is a hybrid composed of generic treenodes and nodes representing semantically rich units such as classes and methods. Actions associated with rules in the Unicon grammar create this hybrid parse tree. Many of the transformations required to translate Unicon into Icon are performed during this phase.

#### 1.1.3.2 Semantic Analysis

The Unicon code that performs the semantic analysis phase of a Unicon-to-Icon translation is a prime example of the power and expressivity of the Unicon language. Semantic analysis of a Unicon unit begins immediately after the parse tree for the unit is populated. The parse tree created during syntax analysis is traversed to identify and perform name mangling on symbols associated with packages and classes. The complexity of this task is compounded by the fact that variables in Unicon (like Icon) need not be declared. Transitive closure of the superclass graph associated with any classes is performed, followed immediately by inheritance resolution for classes. The specification of any package or classes encountered are then written out to a database for subsequent use. Methods associated with objects inserted into the parse tree during syntactic analysis are invoked during semantic analysis to accomplish these tasks.

#### 1.1.3.3 Code Generation

Generating the Icon code for a given Unicon unit is accomplished by a traversal of the parse tree for that unit. Methods associated with objects inserted into the parse tree during syntactic analysis are again used to accomplish code generation. Readability of the final product is facilitated by minor formatting calisthenics performed during this phase.

## 1.2 Motivation

# CHAPTER 2

# UNICONC ORGANIZATION

## 2.1 Topology

Uniconc is composed of three subsystems: a modified version of the Unicon translator (Unicon$^{\Delta}$), a modified version of the Iconc optimizing compiler (Iconc$^{\Delta}$), and a modified version of the Unicon run-time library (RTL$^{\Delta}$). The purpose of each of these subsystems is largely the same as its predecessor, but each has been modified to some degree in order to accomplish the compilation of Unicon programs. The Unicon$^{\Delta}$ subsystem translates programs presented in the Unicon programming language into the Icon programming language. The Iconc$^{\Delta}$ subsystem translates programs presented in the Icon programming language into the C programming language. The RTL$^{\Delta}$ serves to provide functionality commonly used in Unicon and Icon programs to a program executing on a given platform.

The topology of Uniconc as depicted in Figure 2.1 reveals that Uniconc shares some similarities with most traditional compilers while remaining fairly unique. There are three source languages involved in Uniconc. There are effectively four levels of intermediate code in Uniconc. There are two distinct phases of analysis and two distinct phases of synthesis in Uniconc.

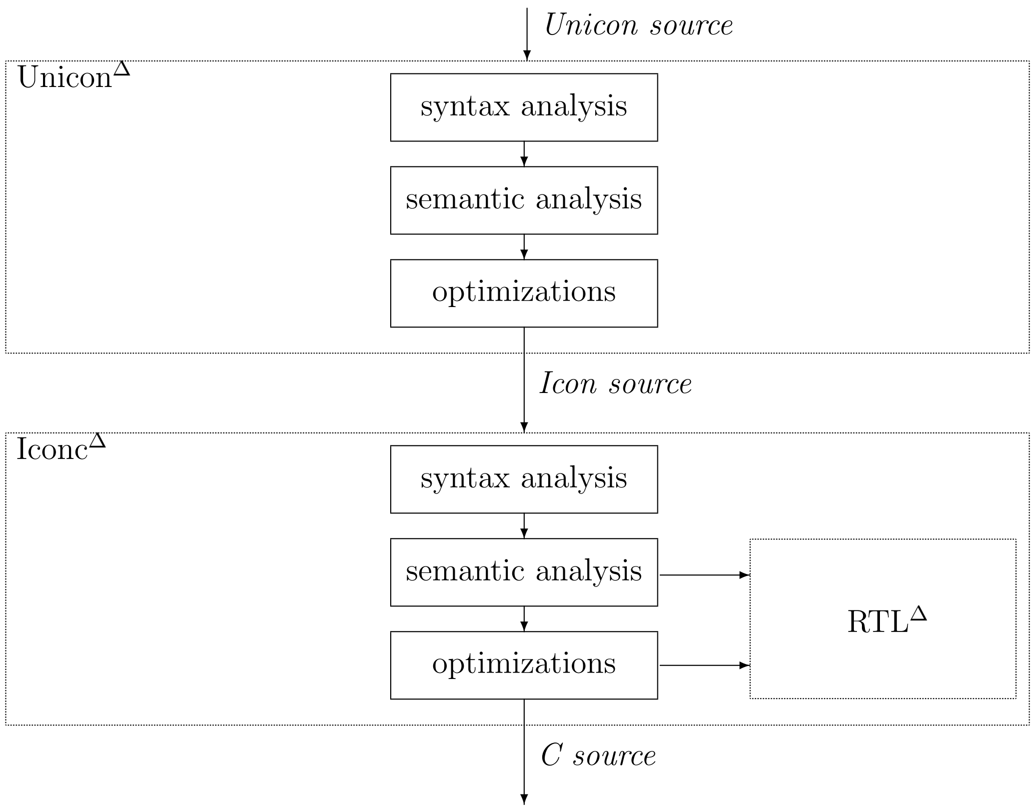

Figure 2.1: Uniconc Topology

## 2.2 Design Considerations

Uniconc is designed to facilitate experimentation. At the outset of this project the decision was made to retain and modify the Unicon translator rather than eliminate it by extending the syntax analyzer for Iconc to include Unicon constructs. This route was the shortest path to achieving the compilation of Unicon programs. More importantly, this decision endowed Uniconc with attributes that make it a useful tool for conducting code transformation experiments.

Intermediate code is produced by Uniconc at four points during the compilation process. Transformations performed in the semantic analysis phase of Unicon$^\Delta$ produce intermediate code that is subsequently transformed by the optimization phase of Unicon$^\Delta$ to produce Icon code acceptable to Iconc$^\Delta$. The Icon code produced by Unicon$^\Delta$ is effectively another level of intermediate code. Transformations performed in the semantic analysis phase of Iconc$^\Delta$ produce intermediate code that is subsequently transformed by the optimization phase of Iconc$^\Delta$ before producing C code that is compiled by the host C compiler of a given system. The C code produced by Iconc$^\Delta$ is effectively another level of intermediate code that is acted upon by the host C compiler. Each of these levels of intermediate code provides a waypoint at which the overall progress of a code transformation experiment can be evaluated. These waypoints also enable the design of code transformation experiments that span one or many levels of intermediate code. The transparency provided by these waypoints decreases the turnaround time for code transformation experiments and thereby increases the time that experimenters can devote to the design and analysis of transformations.

Early experiments with Iconc revealed many impressive features that were clearly of significant value. The decision not to implement a new syntax analyzer for Iconc$^\Delta$ also greatly reduced the number of modifications that were required to produce Iconc$^\Delta$ from Iconc, and thereby reduced the likelihood of introducing errors into an otherwise formidable tool. This same decision increased the number of modifications that were required to produce Unicon$^\Delta$ from Unicon.

Experience has shown, however, that the expressivity and pliability of the Unicon programming language make Unicon$^{\Delta}$ a great place to introduce modifications.

Time, in its varied disguises, is ever a design consideration. During the design of Uniconc, a primary focus was the speed of compiled code. Early efforts in compiling Unicon programs during this project produced unsatisfactory results in this area. The speedup of code compiled by Uniconc was gradually improved by performing code transformation experiments with Uniconc spanning single and multiple segments of intermediate code. The time required to complete code transformation experiments in Uniconc was itself also gradually reduced, but never at the expense of speedup in compiled code. The Uniconc compiler presented in this thesis is the result of many code transformation experiments conducted with earlier versions of Uniconc.

## 2.3 Translation Model

Programming language translation is the heart of any compilation system. Code transformations must be chosen carefully in order accomplish translation that preserves the semantics of programmer input while simultaneously producing the best possible mapping to target code in terms of space-time constraints. The object-oriented Unicon programming language presents many challenges when translating into the procedural Icon programming language. Inheritance mechanisms in the Unicon programming language present particularly acute challenges when translating into the Icon programming language. Namespaces added by the Unicon programming language to provide encapsulation mechanisms also require particular care to ensure that the integrity of a namespace is not violated when translating into a procedural programming language such as Icon. The Unicon programming language has always been a VM-based interpreted language until the advent of Uniconc. Symbol resolution responsibilities that were performed at run-time by the Unicon interpreter require compile-time code transformations and symbol table manipulations that were not previously performed. The goal-directed nature of the Unicon and Icon programming languages poses particularly challenging problems when translating into an imperative programming language such as C. The intricacies involved in translating Icon into C were fortunately solved by the creators of Iconc.

The topology of Uniconc predicates a translation environment that differs from most compilers. Because the ultimate target language of Uniconc is C, traditional code transformational considerations relating to specifics of the target architecture are not a primary concern. This is simultaneously an advantage and a limitation. The peculiarity of generating intermediate code at effectively four places in three forms during the Uniconc compilation process places added emphasis on the order of application of code transformations in Uniconc, and likewise permits experimentation with transformations that span multiple forms or *segments* of intermediate code.

Many of the code transformations that occur during the compilation of a Unicon program by Uniconc are *simple* transformations that are applied within a single segment of intermediate code. Code transformations that are applied across multiple segments of intermediate code in Uniconc are referred to as *compound* or *inter-segment* transformations. The cumulative effect of an inter-segment transformation in Uniconc depends upon a series of subtransformations applied at multiple segments. This phenomenon is referred to as *inter-segment transformational dependency* in Uniconc.

Inter-segment transformations are inherently risky and require extensive experimentation. Care must be taken to ensure that the sequence of subtransformations applied by an inter-segment transformation do not produce side effects that nullify or otherwise dilute the efficacy of other transformations. Analysis of the results produced by an inter-segment transformation can be particularly delicate and time-consuming because the correctness of code is potentially affected at multiple points in the translation stream.

# CHAPTER 3

# THE UNICON$^\Delta$ SUBSYSTEM

## 3.1 Syntax Analysis

A command-line switch was added that permits the Uniconc user to indicate whether the Unicon source code is to be compiled or interpreted. This facility allows Uniconc to generate targets that are interpreted by Iconx or targets that run natively on a targeted platform. The behavior of the Unicon$^\Delta$ syntax analyzer is vastly different if the input source is intended to be compiled. This divergence of behavior is necessary because symbol resolution responsibilities that were performed at run-time by the Unicon interpreter require compile-time code transformations and symbol table manipulations when producing compiled targets. Semantic actions associated with productions in the Unicon programming language grammar were modified to provide distinct funcionality when Uniconc is used as a compiler. Multiple code transformations occur during the Unicon$^\Delta$ syntax analysis phase in the event that Uniconc is being used to produce a compiled result.

The Icon programming language contains a grammatic construct that allows programmers to incorporate library procedures in user-level code. The following Icon expression

```
link strings, graphics
```

directs that the procedures found in the `strings` and `graphics` modules be made available to the Icon source currently being translated. The Unicon programming language inherits this construct from the Icon programming language, and adds a similar grammatic construct known as *packages*. The following Unicon expressions

```
package lang
import gui, cog
```

direct that the classes, declarations, and procedures found in the Unicon source currently being translated be made part of the `lang` package, and that the classes, declarations, and procedures found in the `gui` and `cog` packages be made available to the Unicon source currently being translated. The Unicon expression

```
package f00
```

creates or adds to a namespace `f00`, whereas the expression

```
import f00
```

permits access to the symbols previously defined in the `f00` namespace. This behavior is unlike any construct found in the Icon programming language. The Unicon translator performs a weak link to all packages that are imported, deferring symbol resolution until run-time when the corresponding bytecode is being interpreted. This methodology is unacceptable when attempting to compile Unicon source because Iconc$^\Delta$ requires compile-time symbol resolution in order to perform type inferencing. Uniconc compensates for this by identifying each `link` or `import` target in a compilation and parsing it exactly once. Each new `link` or `import` target encountered in a file that is itself a `link` or `import` target is also parsed. The resolution and parsing of `link` and `import` targets continues until all targets have been parsed once. This problem is solved by defining three tables in Unicon$^\Delta$ called `iconc_links`, `iconc_imports`, and `iconc_parsed`. Each time that a `link` or `import` expression is encountered during the parsing of a Unicon source file, Uniconc resolves the targets associated with the expression. Each target that is not already a member of the `iconc_parsed` table is added to the `iconc_links` or `iconc_imports` table. All members of the `iconc_links` and `iconc_imports` tables are themselves removed from their respective table, parsed, and added to the `iconc_parsed` table. This continues until there are no members remaining in the `iconc_links` and `iconc_imports` tables. The `iconc_links` and `iconc_imports` tables are cleared before the start of parsing each new Unicon source file specified on the Uniconc command-line, whereas the contents of the `iconc_parsed` table persist for the duration of a Uniconc command-line invocation.

Actions associated with productions for invocations in the Unicon grammar have been modified in order to perform code transformations as the parse tree for a Unicon source file is being populated. Different code transformations are performed depending upon the nature of the invocation encountered. The primary motivation for performing these code transformations is to reduce complex expressions in order to simplify vector table resolution in the Iconc$^{\Delta}$ subsystem at compile-time. Rules for invocations appearing in the Unicon programming language grammar are shown in Figure 3.1. The `expr11` nonterminal appearing in Figure 3.1 can produce many grammatic constructs, including other invocations. The `expr11` nonterminal permits Unicon programs to contain arbitrarily complex expressions that describe the entity containing the procedure or method that is to be invoked. A sampling of invocations expressed in the Unicon programming language and the corresponding transformations performed by Uniconc on these invocations appears in Figure 3.2.

Consideration of Figure 3.2 reveals the nature of the transformations performed for invocations. A sequence of transformations are applied to each expression corresponding to an `expr11` until the expression is reduced to an l-value held in a temporary Unicon variable. The type of l-values held in temporary variables is examined during the semantic analysis phase of Iconc$^{\Delta}$ to determine whether further transformations are required in order to reproduce the semantics of the original invocation. In transformation $\mapsto_2$ depicted in Figure 3.2, the type of the l-value `__5` must be ascertained in order to determine whether or not the invocation of `get` is the invocation of a field or a method. If the type of the l-value `__5` is not an instance of a class, no subsequent transformations are required to convey the

```
expr11 :  ...
| expr11 LPAREN exprlist RPAREN {
    $$ := SimpleInvocation($1,$2,$3,$4);
    } ;
| expr11 DOLLAR INITIALLY LPAREN exprlist RPAREN {
    $$ := InvocationNodeShim($1,$2,$3,$4,$5,$6)
    } ;
| expr11 DOLLAR IDENT LPAREN exprlist RPAREN {
    $$ := InvocationNodeShim($1,$2,$3,$4,$5,$6)
    } ;
| expr11 DOLLAR IDENT DOT INITIALLY LPAREN exprlist RPAREN {
    $$ := InvocationNodeShim($1,$2,$3,$4,$5,$6,$7,$8)
    } ;
| expr11 DOLLAR IDENT DOT IDENT LPAREN exprlist RPAREN {
    $$ := InvocationNodeShim($1,$2,$3,$4,$5,$6,$7,$8)
    } ;
```

Figure 3.1: Invocations in the Unicon Grammar

```
x.peek().uncouple(y) ↦1
    (__1 := x.peek(x)) & __1.uncouple(__1,y);

(\n).g.k.get(23) ↦2
    ((__5 := (__4 := (\n).g) & __4.k) & __5.get(23));

self$buf.initially() ↦3
    (self) & (buf__oprec.initially(self));

(\f)$buf.initially() ↦4
    \(f) & (buf__oprec.initially(f));

every (!a)$buf.initially() ↦5
    every ((__2:=(!a))) & (buf__oprec.initially(__2));
```

Figure 3.2: Sample Uniconc Invocation Transformations

semantics of the original invocation. In the event that the type of the l-value `__5` is an instance of a class, another transformation must be applied to the invocation in order to add the implict "self" parameter when invoking the method `get` so that proper state of the class instance `__5` will be maintained and semantic closure can be attained. If `__5` is an instance of a class, the transformation depicted in Figure 3.2 is part of a compound or inter-segment transformation, and the insertion of the implict self argument to attain semantic closure satisfies the inter-segment transformational dependency introduced by the initial transformation depicted in Figure 3.2.

The remaining transformations depicted in Figure 3.2 are simple transformations. In each of these transformations, enough information has been gleaned during the syntax analysis phase of Unicon$^{\Delta}$ to unambiguously determine a single transformation that attains semantic closure with the original invocation. In the case of the invocation of a superclass method, as depicted in transformations $\mapsto_3$, $\mapsto_4$, and $\mapsto_5$ of Figure 3.2, transformations in Unicon$^{\Delta}$ must discard the `$` operator because it is an extension of Icon proper. Earlier versions of Uniconc deferred method resolution until run-time. As experimentation with code transformations in Uniconc progressed, increasing subsets of functionality were added to the compiler and this methodology was discarded where possible in order to produce more efficient run-time behavior of compiled programs. In the rare event that the resolution of a method cannot be accomplished unambiguously at compile-time, Iconc$^{\Delta}$ generates code in the form of a C `switch` statement to achieve resolution of a method at a given code point at run-time.

Actions associated with productions for field references in the Unicon grammar have been modified in order to perform code transformations as the parse tree for a Unicon source file is being populated. Different code transformations are performed depending upon the nature of the field reference encountered. Because a field reference in the Unicon programming language can be produced by the `expr11` nonterminal in the Unicon programming language grammar, chains of field references are reduced to l-values held in temporary Unicon variables. As is the case with method invocations, these l-values are examined during the semantic analysis phase of Iconc$^{\Delta}$ in order to ascertain the type of the entity to which a field reference is being made. The reductions performed by the code transformations applied to field references improve the quality of the code generated by Iconc$^{\Delta}$ by reducing the work necessary to unambiguously determine the parent record or class containing a given named field.

The reductions performed by the code transformations applied to field references also improve the efficiency of compiled targets by ensuring that field references are not unnecessarily evaluated when inserting an implict "self" argument in an invocation that has a field-reference as a constituent `expr11`. In the following code snippet

```
    x := r.o.m(23)
```

the implicit "self" argument that should be added to this invocation is `r.o`. A transformation applied producing this result

```
x := r.o.m(r.o,23)
```

would generate a superfluous second evaluation of `r.o`. A transformation that reduces the expression to an l-value before inserting the implict argument produces

```
x := ((__1 := r.o) & __1.m(__1,23));
```

thereby saving the evaluation of a field reference at run-time. A fundamental problem solved by expression reduction in this case is revealed when attempting to determine the primary nature of `r.o`. In the example depicted above, if `r.o` is not an instance of a class, no transformation is required to achieve semantic closure. Determining whether `r.o` is an instance of a class is occasionally problematic in Unicon$^{\Delta}$. In such cases, Unicon$^{\Delta}$ will produce a transformation of the form

```
x := ((__1 := r.o) & __1.m(23));
```

and defer resolution of the type of `r.o` until after type inferencing is performed during the semantic analysis phase of Iconc$^{\Delta}$. If type inferencing reveals that `__1` (and thereby `o.m`) is an instance of a class, semantic closure of the compound transformation initiated in Unicon$^{\Delta}$ will be achieved by inserting the implicit `__1` as the first argument in the invocation of `m`.

## 3.2 Semantic Analysis

The semantic analysis phase of the current version of Unicon$^{\Delta}$ contains only minor modifications to accomodate Uniconc. Earlier versions of Uniconc contained extensive modifications to the semantic analysis phase of Unicon$^{\Delta}$. Most of the modifications that were present in the Unicon$^{\Delta}$ semantic analysis phase of these earlier versions of Uniconc were mothballed after a system was devised to distinguish between class instances and record instances in Iconc$^{\Delta}$. Before this system was devised, a separate pass over the parse tree during the semantic analysis phase of Unicon$^{\Delta}$ was necessary in order to attempt to disambiguate invocations and field references. The results produced by this additional pass were not of the same caliber that are currently achieved by deferring disambiguation until after type inferencing is performed during the semantic analysis phase of Iconc$^{\Delta}$. The system devised to distinguish between class instances and record instances in Iconc$^{\Delta}$ is described in Section 4.1, and is employed in an optimization described in Chapter 5.

During the semantic analysis phase of Unicon$^{\Delta}$, the parse tree is pruned of all nodes corresponding to `package` and `import` expressions. This pruning is necessary because Iconc$^{\Delta}$ has no knowledge of these grammatic constructs. The parse tree is also pruned of nodes corresponding to `link` expressions because Unicon$^{\Delta}$ handles all such expressions for compiled targets in order to resolve any `package` or `import` expressions that may be present within a file that is itself the target of a `link` expression. Checks are performed during this pruning to ensure that all `package`, `import`, and `link` expressions have been correctly resolved.

## 3.3 Code Generation

Minor modifications to the code generation phase of Unicon$^{\Delta}$ have been made in order to remove tail recursion from Unicon procedures that are recursively invoked while emitting generated Icon code. These modifications were made after exhausting space resources during the code generation phase of Unicon$^{\Delta}$ on some computational platforms. These modifications are active whether a given target is to be interpreted or compiled.

Transformations are applied to Icon code generated by the Unicon$^{\Delta}$ code generation phase in order to produce Icon code that decreases the number of field references required to accomplish the invocation of a Unicon method. This series of transformations is detailed in Section 5.1. The series of transformations performed at this point fundamentally modify the representation of Unicon programming language class instances in Icon code generated by Unicon$^{\Delta}$ and are therefore only performed for targets that are to be compiled. These transformations are accomplished in a separate post-processing pass over the Icon code generated by Unicon$^{\Delta}$. This additional pass imparts visibility into transformational mappings produced, thereby reducing the amount of time required to analyze the results of code transformation experiments performed at this point. As experiments with this series of transformations progresses, these transformations may become part of the Unicon$^{\Delta}$ code generator proper and not require a separate pass over the Icon code generated by Unicon$^{\Delta}$.

# CHAPTER 4

# THE ICONC$^{\Delta}$ SUBSYSTEM

## 4.1 Syntax Analysis

Record types are a primary vehicle for data abstraction in the Icon programming language. The Unicon programming language extends the notion of a record type by permitting users to declare and define (instantiate) classes. Classes in the Unicon programming language are ultimately represented as Icon records before being interpreted or compiled. In certain cases it is very important to be able to differentiate between Icon records that represent Unicon classes and Icon records that represent Icon records. Invocations, as described in Section 3.1, are an example of this necessity. Classes in the Unicon have fundamentally different semantics than records in the Icon. When it is impossible to differentiate between records that represent classes and records that represent records, information and therefore code transformational leverage are lost. In such a case, the designer of code transformations must choose to endow records with the same semantic properties as classes, to demote classes to the semantic equivalent of records, or to defer differentiation until run-time. None of these choices is particularly appealing.

It was recognized early on during the course of this project that the information lost by representing classes as records was information that had to be

```
record class1__state(...)
record class1__methods(...)
global class1__oprec := class1__methods(...)
```

Figure 4.1: Simplified Unicon Class Representation

reclaimed. It was also recognized that requiring the type inferencing phase of Iconc$^{\Delta}$ to perform significantly more computation or to consume more space as a result of this reclamation was not an option. Multiple experiments were performed before settling on an approach that balances the performance requirements of compiled code with the performance requirements of compiling code.

Each Unicon class is translated into a pair of Icon record declarations. These declarations contain a unique signature that can be detected at compile-time. A complete example of the Icon code produced for a Unicon class is shown in Appendix B. A simplified representation of the Icon code produced for a Unicon class is depicted in Figure 4.1. Referring to this simplified representation reveals that a Unicon class named `class1` is translated into a pair of Icon record declarations. The record `class1__state` contains all of the data members associated with an instance of any `class1`, and the singleton instance of `class1__methods` called `class1__oprec` contains the operations shared by any `class1` instances. It should be noted that this is a simplification for the sake of discussing the method that was devised to distinguish between record instances and class instances during the compilation of a Unicon program. The actual representation of Unicon classes is transformed extensively by a post-processor in order to reduce the time required to perform a method invocation in Unicon programs compiled by Uniconc, and to reduce the space consumed by a class instance in Unicon programs compiled by Uniconc.

During the syntax analysis phase of Iconc$^{\Delta}$, each record declaration is added to a list of record entries of type `struct rentry` detailing the record types that are present in a program. Each global entity detected by Iconc$^{\Delta}$ is ascribed a set of attributes or flags that denote the nature of said entity. In the case of a record declaration, the flag `F_Record` is ascribed. Iconc$^{\Delta}$ has been modified to add another attribute, called `F_Object`, that is ascribed in the event that a record declaration conforms to the unique signature indicating that it is the representation of a class instance. This attribute is checked during the semantic analysis and code generation phases of Iconc$^{\Delta}$ in order to perform transformations that increase the efficiency of compiled code.

Further modifications to Iconc$^{\Delta}$ have been made in order to reclaim the semantics of classes. During the syntax analysis phase, Iconc$^{\Delta}$ detects operations records (`xxx__oprec`) instances and creates a virtual table (vtbl) corresponding to each operation record instance encountered. These vtbls are accessed during the semantic analysis and code generation phases of Iconc$^{\Delta}$ in order to reduce the number of field references required to perform a method invocation in compiled Unicon programs. Virtual tables are logical wrappers that encapsulate global

```
ctor := constructor("dbrow", dbcol[1], dbcol[2])
every i := 1 to *dbrows do {
    r := ctor(dbrows[i][1], dbrows[i][2])
    write("row:  ", i, " col[1]:  ", r[1], "col[2]:  ", r[2])
    }
```

Figure 4.2: Dynamic Records in Unicon

entries of type `struct gentry` in Iconc$^{\Delta}$ and provide an interface for membership queries. Each vtbl is a lightweight construct requiring the space of two compile-time pointers.

The Unicon programming language extends the notion of a record type by permitting the instantiation of records whose form is not known until run-time. This language feature known as *dynamic records* is illustrated in Figure 4.2.

This language feature is based upon and exploits run-time information for which there exists no compile-time analog. Uniconc makes a limited effort to determine the number of dynamic records that may be instantiated at run-time. It is necessary for Iconc$^{\Delta}$ to know the number of record types that may be instantiated at run-time in order to efficiently perform type inferencing during the semantic analysis phase. The design of Iconc$^{\Delta}$ predicates that the number of types be known and fixed before type inferencing is performed. The syntax analysis phase of Iconc$^{\Delta}$ has been modified to detect invocations of the `constructor` function and to create new record entries of the form `struct rentry`. Each `struct rentry` created contains the type name of the record and the name and number of the record fields when it is possible to determine this information at compile-time. The number of dynamic records encountered during syntax analysis is tabulated, and this information is used by Iconc$^{\Delta}$ to perform type inferencing. The starting index of dynamic records is embedded in the code generated for a compiled Unicon program. This information embedded in the generated code is used to synchronize the RTL$^{\Delta}$ with the compiled program in order to ensure that the RTL$^{\Delta}$ does not assign record numbers to dynamic records in the `constructor` function that collide with record numbers used for non-dynamic records in a given program. The RTL$^{\Delta}$ has been modified in order to permit the instantiation of dynamic records at run-time in compiled Unicon code. Empirical evidence suggests that the technique to avoid record number collisions at run-time in compiled Unicon programs is effective.

## 4.2 Semantic Analysis

The semantic analysis phase of Iconc$^{\Delta}$ has been modified in order to identify and complete code transformations initiated in Unicon$^{\Delta}$. Each parse tree node representing an invocation is examined during type inferencing in order to identify inter-segment transformational dependencies that remain unsatisfied. Invocation nodes that type inferencing indicates are fields that have the attribute `F_Object` are further examined to determine whether an implicit "self" argument must be added to the argument list of the invocation in order to satisfy an inter-segment

```
case N_Invok:
    /*
    * General invocation.
    */
    infer_nd(Tree1(n)); /* thing being invoked */

    if (Tree1(n)->n_type == N_Field && fldref_is_class(Tree1(n))) {
        methodinvok_add_implicit_self(n);
        }

    /*
    * Perform type inference on all the arguments and copy the
    * results into the argument type array.
    */
    sav_argtyp = arg_typs;
    sav_nargs = num_args;
```

Figure 4.3: Examining Invocations in Iconc$^{\Delta}$

transformational dependency. If an unsatisfied dependency exists, the parse tree node corresponding to the invocation is modified in order to achieve semantic closure. Each modified invocation node is marked to prevent superfluous examinations. The parse tree transformations performed during the semantic analysis phase of Iconc$^{\Delta}$ are simplified by the parse tree transformations performed during the syntax analysis phase of Unicon$^{\Delta}$. Many fruitless experiments were performed before determining an effective method of coordinating inter-segment transformations between Unicon$^{\Delta}$ and Iconc$^{\Delta}$. The C code to examine invocations during type inferencing is depicted in Figure 4.3. The C code to insert an implicit self in a method invocation is depicted in Figure 4.4.

```
static
void
methodinvok_add_implicit_self(n)
    struct node * n;
{
    int i;
    int nargs;
    struct node * t;
    struct node * lhs;

    nargs = Val0(n);
    lhs = Tree0(Tree1(n)); /* lhs of subordinate N_Field */

    if (nargs > 0 && (n->n_field[2].n_ptr->n_col == 123456789 ||
        (n->n_field[2].n_ptr->n_type == lhs->n_type &&
        n->n_field[2].n_ptr->n_field[0].n_ptr ==
        lhs->n_field[0].n_ptr))) {
        /*
        * We have already added the implict self arg to this
        * method call in an earlier typinfer iteration, or it
        * was supplied in the unicon-generated code; move on.
        */
        return;
        }
    t = dupnode(lhs);
    t->n_col = 123456789; /* mark this node as visited */
    i = node_descendants(n);
    n->n_field[nargs+2].n_ptr = NewNode(i);
    for (i=nargs; i>0; i--)
        n->n_field[i+2].n_ptr = n->n_field[i+1].n_ptr;
    Val0(n) += 1;
    n->n_field[2].n_ptr = t;
}
```

Figure 4.4: Implict Self Insertion in Iconc$^{\Delta}$

```
every i := 1 to 5 do {
    ((__1 := (__1:=
    k.nodes_table[i])) &
    (if __1["add_opening"] then __1.add_opening()
    else (__1.__m.add_opening(__1)))) ;
    };
```

Figure 4.5: Subexpression Propagation Example

The model used by Iconc$^{\Delta}$ to evaluate subexpressions during type inferencing has been modified. The necessity of this modification was exposed during the analysis of code transformation experiments that did not provide semantic closure for a specific domain of expressive inputs. Expressions of the form

$e_1$ := $e_2$

were incorrectly evaluated if subexpression $e_2$ contained an assignment to the same l-value represented by subexpression $e_1$. The sequence of types assumed by an Icon variable during a chain of assignments was in some cases not being propagated beyond the first subexpression in which an assignment was made to said variable. This applicative anomaly was perturbed by generating code in Unicon$^{\Delta}$ that aggressively reused temporary variables. An example of code generated by Unicon$^{\Delta}$ that provoked this behavior is shown in Figure 4.5. In this example, the type held in the temporary `__1` was not propagated to the expression containing the ultimate assignment to `__1`, and the code generated by Iconc$^{\Delta}$ for the entire righthand side of the conjunctive was invalid.

Investigation revealed that the applicative order of expression evaluation was being violated during type inferencing in Iconc. The type information associated with the innermost assignment to `__1` in Figure 4.5 was being cleared in order to produce the type information pertaining to the ultimate assignment to `__1`. The immediate workaround for this problem was to implement transformations in Unicon$^\Delta$ that were lax in their use of temporary variables. The semantic analysis phase of Iconc$^\Delta$ was eventually modified to perform an additional check during type inferencing to ensure that type information associated with a variable is not cleared before said information is used to update the type information associated with the same variable.

## 4.3 Code Generation

The code generation phase of Iconc$^\Delta$ has been modified to capitalize on the modifications introduced in Section 4.1 that permit compile-time differentiation between records representing class instances and records representing records. The primary motivation behind these modifications is to increase the efficiency of method invocations in compiled Unicon programs.

Logic has been added inside Iconc$^\Delta$ to examine field references during code generation. Any detected references to fields within records that represent class instances are passed to additional logic that queries the vtbl (virtual table) of the represented class to perform symbol resolution. If a field reference is an invocation and the record containing the field represents a class instance, control is passed to additional logic that generates an invocation of the resolved method. Method invocations are generated using generic Icon descriptors so that the run-time re-assignment of methods will not produce erroneous results.

# CHAPTER 5

# OPTIMIZATIONS IN COMPILED TARGETS

## 5.1 Representation of Class Instances

This section describes a proposed Uniconc optimization that fundamentally modifies the representation of class instances in compiled Unicon targets. This optimization is aggressive and fairly complex. This optimization is primarily time-directed, but also decreases the space required to represent a Unicon class instance in compiled Unicon targets.

### 5.1.1 Background

A series of transformations is performed on Unicon source code to produce Icon source that is translated by the Icon translator Icont, and subsequently executed by the Icon interpreter Iconx. A sample Unicon program for illustrative purposes appears in Appendix A.

Classes are not native to Icont or Iconx. As such, the sample program depicted in Appendix A is translated into a collection of procedures and records recognized as traditional Icon constructs by Icont and Iconx. Transformations pertaining to inheritance, method resolution, etc., are performed on the Unicon source to produce semantically equivalent procedural code acceptable to Icont.

The code generated by Unicon for the sample program depicted in Appendix A is shown in Appendix B.

It can be seen in Appendix B that the current code generation model produces two record declarations to represent the `fifo` class. The `record fifo__methods` contains a field for each member method of a `fifo` class. A single instance of a `fifo__methods` record is created at run-time and shared by all `fifo` objects. Each `fifo` instance refers to this methods vector through the `__m` field of its corresponding `record fifo__state`. One `fifo__state` record instance is created for each run-time `fifo` object. The `fifo__state` record contains fields for all member variables in a `fifo` instance, a field for the implicit `self` member specific to object-oriented programming languages, and the aforementioned methods vector `__m`.

A code snippet from Appendix B illustrating the internal representation of a Unicon object in the procedural realm of Icon appears in Figure 5.1. The `procedure fifo` represents the constructor of a `fifo` object. The first call to this constructor instantiates a `fifo__methods` record named `fifo__oprec` containing all procedures representing the methods contained in the `fifo` class. The `procedure fifo` creates an instance of the `fifo__state` record, populates the `__m` and `__s` fields with the methods vector and self reference, respectively, and invokes the initially procedure corresponding to the `procedure fifo_initially` to initialize the explicit member variables of this `fifo` instance.

```
record fifo__state(__s,__m,m_data)
record fifo__methods(get,peek,put,size,initially,buf)
global fifo__oprec, buf__oprec
procedure fifo()
local self,clone
initial {
    if /fifo__oprec then fifoinitialize()
    if /buf__oprec then bufinitialize()
    fifo__oprec.buf := buf__oprec
    }
    self := fifo__state(&null,fifo__oprec)
    self.__s := self
    self.__m.initially(self,) | fail
    return self
end

procedure fifoinitialize()
    initial fifo__oprec := fifo__methods(fifo_get,fifo_peek,fifo_put,
        fifo_size,fifo_initially)
end
```

Figure 5.1: Unicon Object as Procedural Entity

A variable in the Unicon or Icon programming language may assume many types over its lifetime, and variables in these languages need not be declared before their use. This flexibility contributes admirably to the rapid development model for which these languages are renowned, and is simultaneously a source of consternation when attempting to translate down to a high level language with a more rigid type system. Iconc employs a type inferencing system based upon global data flow analysis to determine the types that any variable may take during specific points of program execution. Empirical data suggests that this type inferencer is highly effective, particularly because Icon programs typically make extensive use of global variables. The code generation model of Unicon uses global variables to represent class-wide data shared among Unicon class instances.

Records are used to implement abstract data types in Icon, and Unicon classes are themselves represented as Icon records. Because Unicon is an object-oriented programming language, it provides mechanisms for the encapsulation of data and operations within an object or l-value, and permits the access or mutation of objects through field references associated with the object to be manipulated. The typing system used by the Iconc type inferencer treats records with particular rigor. Each Icon record is considered a distinct type, and each field within each record is itself a distinct type. This treatment of records contributes to the thorough analysis of the flow of data among Icon variables and facilitates the generation of efficient C code to represent Icon expressions.

```
struct fentry { /* field table entry */
    struct fentry *blink;   /* link for bucket chain */
    char *name;             /* name of field */
    struct par_rec *rlist;  /* head of list of records */
    };
```

Figure 5.2: Field Representation in Iconc

```
struct par_rec { /* list of parent records for a field name */
    struct rentry *rec; /* parent record */
    int offset;         /* field's offset within this record */
    int mark;           /* used during code generation */
    struct par_rec *next;
    };
```

Figure 5.3: Parent Record Representation in Iconc

When a field reference is encountered during the parsing of Icon code, Iconc creates a node in its internal parse tree corresponding to the field reference. Iconc instantiates a `struct fentry` of the form seen in Figure 5.2 for each field encountered. The declaration of a `struct fentry` reveals that Iconc associates a list of `struct par_rec` with each field entry. The `struct par_rec` list represents all records containing a field of a given name. The declaration of `struct par_rec` appears in Figure 5.3.

All `struct fentry` instances are hashed using the name of the field as the hashing key. The `struct par_rec` declaration contains the field offset for the field of a given name within a particular parent record. When a field reference

```
{
/* lkup undo__UndoableEdit */
struct b_record *r_rp = (struct b_record *)
    BlkLoc(r_f.t.d[0] /* self */);
r_f.t.d[2].dword = D_Var +
    ((word *)&r_rp->fields[8] - (word *)r_rp);
VarLoc(r_f.t.d[2]) = (dptr)r_rp;
}
```

Figure 5.4: Iconc Code Generated for Unambiguous Field Reference

is encountered in the parse tree during type inferencing, Iconc infers the types that the record may assume at the location of the field reference during program execution. Iconc then uses the name of the field to which the code refers to determine the parent records associated with the field and the offsets of the field within said parent records. This information is used in the event that a lefthand side of a field reference may assume multiple types at a given point in program execution. The information regarding the types that the field may assume during program execution is stored in a specific area of the parse tree node representing the field reference.

If the type of a record can be determined unambiguously at compile-time, Iconc generates code that directly accesses a referenced field in the record. The code that Iconc generates in this case is of the form seen in Figure 5.4. In this case, the Iconc type inferencer has determined that the lefthand side of the field reference at this code point can only take one type during program execution, and Iconc has determined the offset of the field to be accessed by examining the `struct par_rec` corresponding to the type of the record.

Often the Iconc type inferencer determines that the lefthand side of a field reference at a given code point may assume more than one record type during program execution. In this case, Iconc examines all records associated with the types that the lefthand side may assume and, if only a single record contains a field of the name specified at that code point or if the field to be referenced is at the same offset within all records containing the field, Iconc is still able to generate code that is of the form seen in Figure 5.4.

In many cases, however, more than one record contains a field of the name specified in the field reference, and the offsets for the named field within one or more of these records is not the same. In this case, Iconc generates a C switch statement to perform a run-time check to determine the type of the record being accessed and the offset of the field within that record. The code generated by Iconc in this instance is of the form shown in Figure 5.5.

In this case Iconc generates code to examine the record number of the lefthand side of the field reference and assigns the correct field offset based upon the number (type) of the record being accessed. This information is gathered from the `struct fentry` and `struct par_rec` lists at compile-time.

```
{
/* lkup MaxChars */
struct b_record *r_rp = (struct b_record *)
    BlkLoc(glbl_argp[0] /* self */);
dptr r_dp; int fld_idx;
switch (r_rp->recdesc->proc.recnum) {
    case 127:
        r_dp = &r_rp->fields[4];
        break;
    case 285:
        r_dp = &r_rp->fields[1];
        break;
    default:
        if ((fld_idx = fldlookup(r_rp, "MaxChars")) < 0)
            err_msg(207, &glbl_argp[0] /* self */);
        else
            r_dp = &r_rp->fields[fld_idx];
    }
r_f.t.d[11].dword = D_Var + ((word *)r_dp - (word *)r_rp);
VarLoc(r_f.t.d[11]) = (dptr)r_rp;
}
```

Figure 5.5: Iconc Code Generated for Ambiguous Field Reference

### 5.1.2 The Proposed Optimization

This optimization originated as a time-directed optimization during the experimentation with and analysis of C code generated for Unicon source. Since Unicon is object-oriented, an obvious path to faster generated code seemed to be through the tailoring of field references. The reader will recall that a method reference in Unicon source is actually presented as a pair of field references to Iconc. The first reference is to obtain the `__m` field (methods vector) of the object, and the second is to access the desired field containing the desired method. The primary goal at the outset of this optimization work was to eliminate this extra field reference in compiled Unicon targets.

The first step toward eliminating the extra field reference for method invocations in compiled Unicon targets was to modify the Unicon code generation model. Unicon$^{\Delta}$ was modified to eliminate all references to `__m` and `__s` in generated code targeted for Iconc$^{\Delta}$. The translation of a Unicon class into Icon records was modified to facilitate the removal of `__m` and `__s`. The code generated by Unicon$^{\Delta}$ for the sample program depicted in Appendix A now appears as shown in Appendix C.

Comparing the generated code in Appendix C with the generated code depicted in Appendix B reveals several points of interest worthy of discussion. The `record fifo__methods` declaration and its instance `global fifo__oprec` are the same in both of these appendices. This is due to the fact that the proposed optimization retains the previous behavior of using a single, shared methods vector among all instances of a Unicon class. The previous representation of the `fifo` class in Appendix B

```
record fifo__state(__s,__m,m_data)
record fifo__methods(get,peek,put,size,initially,buf)
```

now appears as

```
record fifo__mdw_inst_mdw(m_data,get,peek,put,size,initially,buf)
record fifo__methods(get,peek,put,size,initially,buf)
```

in the code generated by Unicon$^{\Delta}$ as modified for this proposed optimization. The reader will immediately note that the `fifo__mdw_inst_mdw` declaration, the analog of the `fifo__state` declaration under the previous translation model, has more fields than the previous `fifo__state`, and that the `record fifo__mdw_inst_mdw` declaration contains fields that are redundant with those contained in the `record fifo__methods` declaration. This is intended.

The field redundancy between the `fifo__methods` and `fifo__mdw_inst_mdw` records under the proposed optimization is a bit of trickery directed at the Iconc$^{\Delta}$ parser. This field redundancy leads Iconc$^{\Delta}$ to permit the notation `r.add()` *invocation P*

where

`r.__m.add()` *invocation O*

was previously required. When Iconc$^{\Delta}$ parses invocation P, it determines that invocation P is syntactically correct due to this intentional field redundancy. After type inferencing is complete, the redundant fields in the `struct rentry` of the `struct par_rec` that Iconc$^{\Delta}$ builds to represent the `struct fifo__mdw_inst_mdw` are removed. This removal makes these fields and their offsets inaccessible to the Iconc$^{\Delta}$ code generator. The removal of redundant fields from Iconc records representing Unicon class instances is accomplished by the source code depicted in Figure 5.6.

All field references are examined during the code generation phase of Iconc$^{\Delta}$. If a given field reference is an invocation and the type of the field is of one or more Unicon class record instances, control is passed to additional logic added inside Iconc$^{\Delta}$ to resolve the particular vector table to which the reference is directed.

### 5.1.3 Results

The reader will recall that the code to add a datum to a `fifo` as it appears in the sample program in Appendix A is `f.put("1st")` as shown in the `procedure fifo_test`. A snippet of code generated by Iconc to perform the above invocation

```
static
void
adjust_class_recs(recs)
    struct rentry * recs;
{
    int nflds;
    char * p, * q;
    struct fldname * f;
    struct rentry * rinst;
    struct rentry * rmeth;

    for (rinst=recs; rinst; rinst=rinst->next) {
        if ((p = strstr(rinst->name, "__mdw_inst_mdw")) == NULL)
            continue;
        for (rmeth=rinst->next; rmeth; rmeth=rmeth->next) {
            if ((q = strstr(rmeth->name, "__methods")) == NULL)
                continue;
            if (p - rinst->name != q - rmeth->name)
                continue;
            if (strncmp(rinst->name, rmeth->name, p - rinst->name))
                continue;
            nflds = rinst->nfields - rmeth->nfields;
            while (rinst->nfields > nflds) {
                f = rinst->fields;
                rinst->fields = rinst->fields->next;
                free(f);
                rinst->nfields--;
                }
            break;
            }
        }
}
```

Figure 5.6: Record Field Removal in Iconc$^{\Delta}$

using the methodology existing prior to this proposed optimization appears in Figure 5.7. Under this proposed optimization, Iconc$^{\Delta}$ generates code to perform this same invocation as shown in Figure 5.8.

Consideration of Figure 5.7 and Figure 5.8 reveals that an extra field reference is indeed saved using the proposed optimization. The savings in time realized by this optimization will be proportional to the number of method invocations executed by a compiled Unicon program. Another effect of the proposed optimization is the decrease of run-time space required to represent a Unicon object in code generated by Iconc$^{\Delta}$. The space required to represent two pointers in compiled Unicon targets is eliminated for each run-time instance of a Unicon class under the proposed optimization because the fields `__m` and `__s` are no longer used. This side effect will likely prove beneficial in large-scale object-oriented applications where Unicon is typically deployed.

## 5.2 Invocations

This section describes a proposed Uniconc optimization that modifies numeric parameters passed to invocations in compiled Unicon and Icon targets. This optimization is primarily time-directed.

### 5.2.1 Background

The RTL$^{\Delta}$ contains a generalized invocation function called `invoke`. Code generated by Iconc and Iconc$^{\Delta}$ often invokes `invoke` in order to accomplish the

```
L30:  ; /* is record */
    {
    struct b_record *r_rp = (struct b_record *)
        BlkLoc(r_f.t.d[0]/* f */);
    /* mdw:  collapsed __m switch */
    r_f.t.d[3].dword = D_Var +
        ((word *)&r_rp->fields[1] - (word *)r_rp);
    VarLoc(r_f.t.d[3]) = (dptr)r_rp;
    }
    deref(&r_f.t.d[3], &r_f.t.d[3]);
    if ((r_f.t.d[3]).dword == D_Record)
        goto L31 /* is record */;
    err_msg(107, &r_f.t.d[3]);
L31:  ; /* is record */
    {
    struct b_record *r_rp = (struct b_record *) BlkLoc(r_f.t.d[3]);
    dptr r_dp; int fld_idx;
    switch (r_rp->recdesc->proc.recnum) {
        case 1:
        case 3:
        case 5:
            r_dp = &r_rp->fields[2];
            break;
        default:
            if ((fld_idx = fldlookup(r_rp, "put")) < 0)
                err_msg(207, &r_f.t.d[3]);
            else
                r_dp = &r_rp->fields[fld_idx];
        }
    r_f.t.d[2].dword = D_Var + ((word *)r_dp - (word *)r_rp);
    VarLoc(r_f.t.d[2]) = (dptr)r_rp;
    }
    r_f.t.d[3].dword = D_Var;
    r_f.t.d[3].vword.descptr = &r_f.t.d[0] /* f */;
    r_f.t.d[4].vword.sptr = "1st";
    r_f.t.d[4].dword = 3;
    invoke(3, &r_f.t.d[2], &r_f.t.d[5], sig_28);
L29:  ; /* bound */
```

Figure 5.7: Sample Method Invocation in Iconc

```
L12:  ; /* is record */
    {
    /* mi:  lkup put in fifo__oprec */
    struct b_record *r_rp = (struct b_record *)
        BlkLoc(globals[17]);
    r_f.t.d[2].dword = D_Var +
        ((word *)&r_rp->fields[2] - (word *)r_rp);
    VarLoc(r_f.t.d[2]) = (dptr)r_rp;
    }
    r_f.t.d[3].dword = D_Var;
    r_f.t.d[3].vword.descptr = &r_f.t.d[0] /* f */;
    r_f.t.d[4].vword.sptr = "1st";
    r_f.t.d[4].dword = 3;
    invoke(3, &r_f.t.d[2], &r_f.t.d[5], sig_13);
```

Figure 5.8: Sample Method Invocation in Iconc$^{\Delta}$

```
int invoke(nargs, args, rslt, succ_cont)
    int nargs;
    dptr args;
    dptr rslt;
    continuation succ_cont;
```

Figure 5.9: Signature of `invoke` Function

invocation of a procedure or function represented by a procedure descriptor or a string descriptor. The signature of `invoke` appears in the RTL$^{\Delta}$ as shown in Figure 5.9. The `dptr` type shown in Figure 5.9 is a pointer to a descriptor. Descriptors are used in the RTL$^{\Delta}$ and the Icon VM to generically describe the type and value of Icon and Unicon run-time entities. A typical invocation of `invoke` before acted upon by this proposed optimization is shown in Figure 5.10.

```
    r_f.t.d[6].dword = D_Integer;
    r_f.t.d[6].vword.integr = 3;
    r_f.t.d[3].dword = D_Var;
    r_f.t.d[3].vword.descptr = &r_f.t.d[0] /* f */;
    r_f.t.d[4].vword.sptr = "1st";
    r_f.t.d[4].dword = 3;
    invoke(r_f.t.d[6].vword.integr, &r_f.t.d[2], &r_f.t.d[5],
        sig_28);
L29:  ; /* bound */
```

Figure 5.10: Sample Invocation in Uniconc Before Optimization

### 5.2.2 The Proposed Optimization

This optimization adds logic to Iconc$^{\Delta}$ in order to determine at compile-time the value held by the integer descriptor that is declared as the formal argument `nargs` for each invocation of the `invoke` function. This optimization is enabled by a Uniconc command-line option. The logic added to Iconc$^{\Delta}$ by this optimization creates a temporary compile-time descriptor which is marked to be accessed as an integer literal containing the value of the integer descriptor that would normally be used to accomplish the invocation. This temporary descriptor is used by the Iconc$^{\Delta}$ code generator to populate the codestream with the integer literal itself rather than the sequence of field references that are normally used to access the integer literal within the given descriptor. Figure 5.11 depicts the result of this proposed optimization. The `invoke` function being used to accomplish the invocation of an Icon procedure in a compiled target shown at the end of Figure 5.11 reveals that the logic added to Iconc$^{\Delta}$ by this proposed optimization indeed in-

```
    r_f.t.d[3].dword = D_Var;
    r_f.t.d[3].vword.descptr = &r_f.t.d[0] /* f */;
    r_f.t.d[4].vword.sptr = "1st";
    r_f.t.d[4].dword = 3;
    invoke(3, &r_f.t.d[2], &r_f.t.d[5], sig_28);
L29:  ; /* bound */
```

Figure 5.11: Sample Invocation in Uniconc After Optimization

serts an integer literal in the codestream where previously there were a sequence of field references. The logic added to Iconc$^{\Delta}$ that accomplishes this substitution is shown in Figure 5.12.

### 5.2.3 Results

Consideration of Figure 5.10 and Figure 5.11 indicates that eleven field references, three array subscript operations, and two assignment operations are eliminated at run-time for each invocation of `invoke` acted upon by this proposed optimization. This proposed optimization only acts upon those invocations that are not already correctly inlined by Iconc, so the aforementioned benefit is realized if and only if the previously existing Iconc code failed to reduce the descriptor to an integer literal.

## 5.3 Dereferences

This section describes a proposed Uniconc optimization that generates inline C code in certain cases where a call to the `deref` RTL$^{\Delta}$ function would normally be generated in compiled Unicon and Icon targets. This optimization is

```
static
void
sub_ilc_fncall_explicit_arg(argilc, protoilc, cd, indx)
    struct il_c * argilc;
    struct il_c * protoilc;
    struct code * cd;
    int indx;
{
    int loctype;

    for (; argilc && protoilc; argilc=argilc->next,
        protoilc=protoilc->next) {
        if (argilc->il_c_type != ILC_Ref) {
            /* process nonmodifying arg references only */
            sub_ilc(argilc, cd, indx);
            continue;
        }
        loctype = cur_symtab[argilc->n].loc->loc_type;
        if (loctype != V_Temp && loctype != V_NamedVar) {
            /* process args for temp locs only */
            sub_ilc(argilc, cd, indx);
            continue;
        }
        if (protoilc->s == NULL ||
            strncmp("C_integer", protoilc->s, 9)) {
            /* currently process only C_integer type */
            sub_ilc(argilc, cd, indx);
            continue;
        }
        cd->ElemTyp(indx) = A_ValLoc;
        cd->ValLoc(indx) = loc_cpy(cur_symtab[argilc->n].loc,
            M_CInt);
    }
}
```

Figure 5.12: Inlining Integer Arguments in Iconc$^\Delta$

primarily time-directed.

### 5.3.1 Background

Variables in Icon and Unicon may assume many types during the lifetime of a program. The implementors of the Icon interpreter developed the notion of a descriptor to generically describe the type and value of a given Icon variable. This method of description is also used in the Icon run-time library, and was subsequently adopted by Iconc. The declaration of a descriptor is shown in Figure 1.1. Performance analysis of compiled targets using profiling tools indicates that the typical Icon or Unicon program spends an appreciable amount of time dereferencing descriptors in the RTL$^{\triangle}$ `deref` function.

### 5.3.2 The Proposed Optimization

When the types of the operands to the `deref` operation can be reliably determined at compile-time, there are situations where the functionality provided by the `deref` function can be generated directly in the codestream without entering the RTL$^{\triangle}$. In the event that the entity being dereferenced is not a variable, or is a "normal" variable, the code contained in the `deref` function is simple and should provide an opportunity for inlining.

Iconc has been modified to examine the operands of the `deref` operation and to generate the inline equivalent of a `deref` call where appropriate. This optimization is enabled by a Uniconc command-line option. This optimization

```
deref(&r_f.t.d[5], &r_f.t.d[5]);
```

Figure 5.13: Code Generated Before `deref` Optimization

is intended to act only in the case where the dereferenced entity is a "simple" variable, or is not a variable at all.

### 5.3.3 Results

A snippet of code generated by Iconc$^\triangle$ without the proposed optimization is shown in Figure 5.13. In this snippet, the result of `O114_subsc` is used as both the source and destination of a dereferencing operation via the RTL$^\triangle$ `deref` function.

A snippet of code generated by Iconc$^\triangle$ for the same source with the proposed optimization enabled is shown in Figure 5.14. The call to `deref` in Figure 5.13 has been replaced with the inlined equivalent of a `deref` call in Figure 5.14. It should be noted that `VarLoc` and `Offset` are macros in the generated code, so no function call is therefore being made by the inlined substitute for the call to the `deref` function in the RTL$^\triangle$.

It is likely that the methodology employed by this particular optimization to circumvent a call to an RTL$^\triangle$ operation can be applied to other RTL$^\triangle$ operations in a similar fashion. Applying this methodology to other RTL$^\triangle$ functions is a potential subject of future experimentation.

```
r_f.t.d[5] = *(dptr)((word *)VarLoc(r_f.t.d[5]) +
    Offset(r_f.t.d[5]));
```

Figure 5.14: Code Generated After `deref` Optimization

# CHAPTER 6

# METRICS

## 6.1 Overview

Uniconc can produce compiled or interpreted Unicon or Icon targets. The purpose of this chapter is to quantify the speedup provided by Uniconc to compiled versus interpreted Unicon code. This speedup is compared to the speedup provided by Uniconc to compiled versus interpreted Icon code. A representative set of programs were chosen to catalogue the important functional features of Icon and Unicon while providing a comparative analysis of these features. The set of programs contained herein is by no means exhaustive. All measurements were taken on an AMD64 dual-core dual-processor machine running linux (Fedora Core 3). Each processor has a clock speed of 1790.9 MHz and has 1MB cache.

## 6.2 Invocations

Invocations are a cornerstone of any programming language. Difficulties arise when attempting to compare invocations in procedural languages to invocations in object-oriented languages due to the fact that invocations in object-oriented languages are fundamentally different than invocations in procedural languages.

```
procedure p()
    return;
end

procedure main(argv)
    local n, clk;

    n := integer(argv[1]);
    clk := &time;
    while (n > 0) do {
        p();
        n -:= 1;
        }
    clk := &time - clk;
    write("time:  ", clk);
end
```

Figure 6.1: Icon Program Measuring Invocations

Figure 6.1 is an Icon program used to measure the speed of invocations in compiled and interpreted Icon targets. Figure 6.2 contains a Unicon program used to measure the speed of invocations in compiled and interpreted Unicon targets. All targets, compiled and interpreted, were generated by the same version of Uniconc. Each target was invoked with a single command-line argument in order to accomplish 10,000,000 invocations. The measurements depicted are the average of three runs of each program. The results are shown below.

| Target | average time (ms) | | |
|---|---|---|---|
| compiled Icon | 703 | | |
| interpreted Icon | 8,560 | | |
| | | Icon speedup | **1,217%** |
| compiled Unicon | 1,190 | | |
| interpreted Unicon | 10,150 | | |
| | | Unicon speedup | **853%** |

```
class o()
    method p()
        return;
    end
end

procedure main(argv)
    local n, x, clk;

    x := o();
    n := integer(argv[1]);
    clk := &time;
    while (n > 0) do {
        x.p();
        n -:= 1;
        }
    clk := &time - clk;
    write("time:  ", clk);
end
```

Figure 6.2: Unicon Program Measuring Invocations

```
procedure tak(x, y, z)
    if not (y < x) then
        return z
    else
        return tak(tak(x-1, y, z), tak(y-1, z, x), tak(z-1, x, y));
end

procedure main(argv)
    tak(integer(argv[1]), integer(argv[2]), integer(argv[3]));
end
```

Figure 6.3: Icon Takeuchi Function

Figure 6.3 is an Icon program using the Takeuchi function to measure the speed of recursive invocations in compiled and interpreted Icon targets. Figure 6.4 contains a Unicon program using the Takeuchi function to measure the speed of recursive invocations in compiled and interpreted Unicon targets. All targets, compiled and interpreted, were generated by the same version of Uniconc. Each target was invoked with a single command-line argument of the form `tak 27 18 9`. The measurements depicted are the average of three runs of each program taken with the linux `time(1)` utility. The results are shown below.

| Target | average time (ms) | | |
|---|---|---|---|
| compiled Icon | 500 | | |
| interpreted Icon | 11,400 | | |
| | | Icon speedup | **2,280%** |
| compiled Unicon | 2,070 | | |
| interpreted Unicon | 13,600 | | |
| | | Unicon speedup | **657%** |

It is apparent that the speedup afforded compiled Unicon invocations by

```
class tak()
    method comp(x, y, z)
        if not (y < x) then
            return z
        else
            return comp(comp(x-1, y, z), comp(y-1, z, x),
                comp(z-1, x, y));
    end
initially(x, y, z)
    comp(x, y, z);
    return;
end

procedure main(argv)
    local t;
    t := tak(integer(argv[1]), integer(argv[2]), integer(argv[3]));
end
```

Figure 6.4: Unicon Takeuchi Function

Uniconc is not on par with the speedup afforded compiled Icon invocations by Uniconc. The results are not discouraging, but certainly indicate that further research in this area is necessary.

## 6.3 Field References

Field references are a mainstay of object-oriented programs. The speed at which a field reference is accomplished is of paramount concern when evaluating the utility of an object-oriented language.

Figure 6.5 is an Icon program used to measure the speed of field references in compiled and interpreted Icon targets. Figure 6.6 contains a Unicon program

```
record rec(x)

procedure main(argv)
    local n, r, clk;

    n := integer(argv[1]);
    r := rec(11);
    clk := &time;
    while (n > 0) do {
        v := r.x;
        n -:= 1;
        }
    clk := &time - clk;
    write("time:  ", clk);
end
```

Figure 6.5: Icon Program Measuring Field References

used to measure the speed of field references in compiled and interpreted Unicon targets. All targets, compiled and interpreted, were generated by the same version of Uniconc. Each target was invoked with a single command-line argument in order to accomplish 10,000,000 field references. The measurements depicted are the average of three runs of each program. The results are shown below.

| Target | average time (ms) | | |
|---|---|---|---|
| compiled Icon | 760 | | |
| interpreted Icon | 8,500 | | |
| | | Icon speedup | **1118**% |
| compiled Unicon | 760 | | |
| interpreted Unicon | 8,610 | | |
| | | Unicon speedup | **1133**% |

The results indicate that the speedup afforded compiled Unicon field references by Uniconc is similar to the speedup afforded compiled Icon field references

```
class o(m_x)
    method run(n)
        local v, clk;
        clk := &time;
        while (n > 0) do {
            v := self.m_x;
            n -:= 1;
            }
        clk := &time - clk;
        write("time:  ", clk);
    end
initially(n)
    run(n);
end

procedure main(argv)
    local n, r;

    n := integer(argv[1]);
    r := o(n);
end
```

Figure 6.6: Unicon Program Measuring Field References

by Uniconc.

## 6.4 General Programs

The quicksort algorithm was selected as a benchmark primarily because of its ubiquity. It is also recursive, and sorts in place so the number of allocations is likely reduced. Figure 6.7 shows the Icon implementation of quicksort used for this measurement, and Figure 6.8 shows the Unicon implementation of quicksort used. All targets, compiled and interpreted, were generated by the same version of Uniconc. Each target was invoked with a single command-line argument in order to accomplish the sorting of 200,000 pseudorandom values. The measurements depicted are the average of three runs of each program taken with the linux `time(1)` utility.

| Target | average time (ms) | | |
|---|---|---|---|
| compiled Icon | 1,090 | | |
| interpreted Icon | 4,790 | | |
| | | Icon speedup | **440%** |
| compiled Unicon | 1,260 | | |
| interpreted Unicon | 6,090 | | |
| | | Unicon speedup | **483%** |

The seed used by the pseudorandom number generation facility in Icon and Unicon is the same for each program, and the values being sorted by the programs are therefore the same. The results indicate that the speedup afforded compiled Unicon by Uniconc for such a sorting program is comparable to the speedup afforded compiled Icon by Uniconc.

```
procedure partition(a, p, r)
    local x, i, j;

    x := a[r];
    i := p - 1;
    every j := p to r-1 do {
        if (a[j] <= x) then {
            i +:= 1;
            a[i] :=:  a[j];
            }
        }
    a[i+1] :=:  a[r];
    return i+1;
end

procedure quicksort(a, p, r)
    if (p < r) then {
        q := partition(a, p, r);
        quicksort(a, p, q-1);
        quicksort(a, q+1, r);
        }
end

procedure main(args)
    local a, i, n;

    a := list();
    n := (integer(\args[1]) | 32768);
    every i := 1 to n do
        put(a, ?1048576);
    quicksort(a, 1, *a);
end
```

Figure 6.7: Icon Quicksort Program

```
class sorter(m_a)
    method partition(p, r)
        x := m_a[r];
        i := p - 1;
        every j := p to r-1 do {
            if (m_a[j] <= x) then {
                i +:= 1;
                m_a[i] :=:  m_a[j];
                }
            }
        m_a[i+1] :=:  m_a[r];
        return i+1;
    end
    method quicksort(p, r)
        if (p < r) then {
            q := partition(p, r);
            quicksort(p, q-1);
            quicksort(q+1, r);
            }
        return;
    end
initially(n)
    m_a := list();
    /n := 32768;
    every i := 1 to n do
        put(m_a, ?1048576);
    quicksort(1, *m_a);
    return;
end

procedure main(args)
    local inst;
    inst := sorter(args[1]);
end
```

Figure 6.8: Unicon Quicksort Program

Appendix E contains the listing for an I/O-intensive Unicon program that creates an arbitrary number of Unix resource files containing an arbitrary number of topics in each file. Appendix D contains the listing for the Icon counterpart to the aforementioned Unicon program. All targets, compiled and interpreted, were generated by the same version of Uniconc. Each target was invoked with command-line arguments to accomplish the creation of 50 resource files, with each file containing 500 topics. The filesystem was cleared of resource files created for a given program run after each run was measured. A new class instance (in Unicon) or record instance (in Icon) is created for each resource file. The measurements depicted are the average of three runs of each program taken with the linux `time(1)` utility.

| Target | average time (ms) | | |
|---|---|---|---|
| compiled Icon | 1,030 | | |
| interpreted Icon | 4,960 | | |
| | | Icon speedup | **481**% |
| compiled Unicon | 1,080 | | |
| interpreted Unicon | 5,060 | | |
| | | Unicon speedup | **468**% |

# CHAPTER 7

# RELATED WORK

# CHAPTER 8

# FUTURE WORK

## 8.1 Limitations

### 8.1.1 Scalability

Uniconc currently has a scalability issue that prohibits the compilation of large programs with compuatational resources available to the typical user. Sources of this scalability issue are:

• The parse tree for an entire program is maintained in memory by Iconc$^{\Delta}$ for the duration of compilation.

• The representation of type vectors used by Iconc$^{\Delta}$ can grow very large when many distinct types are used within a program. This is particularly evident in Unicon programs where many classes (records) are used.

• The representation of type vectors used by Iconc$^{\Delta}$ can become pathologically sparse when many distinct types are used within a program. This sparseness tends to increase as type inferencing progresses.

Experiments to improve the scalability of Uniconc during the course of this project have thus far proven unsuccessful. The preliminary underpinnings of further Uniconc scalability experiments are currently being planned. This issue will likely be a primary focus of continuing Uniconc reasearch.

```
class f00(x)
    method reassign_member_var(p)
        x := p;
        x();
    end
end
```

Figure 8.1: Unicon Member Variable Reassignment

### 8.1.2 Class Member Reassignments

The Iconc$^{\Delta}$ code generator currently does not correctly differentiate between methods and procedures in the case where the member variable of a class instance is reassigned with a procedure. Figure 8.1 depicts a situation where the Iconc$^{\Delta}$ code generator currently fails. In particular, the code generated for the invocation of the member variable `x` in `class f00` shown in Figure 8.1 will be invalid. This is because the lefthand side of the generated `self.x()` invocation is a class, so an implicit self argument will be added to the invocation by the Iconc$^{\Delta}$ code generator. A potential avenue of approaching the solution to this error has been devised, but the experimentation required to achieve a solution has not yet been accomplished.

# CHAPTER 9

# CONCLUSION

# APPENDICES

# APPENDIX A

# SAMPLE UNICON PROGRAM

```
class buf(m_data)
    abstract method get()
    abstract method peek()
    abstract method put()
    abstract method size()
initially(val, cnt)
    self.m_data := list()
    if (\cnt) then {
        every i := 1 to cnt do
        put(m_data, val)
        }
    else if (\val) then
        put(m_data, val)
end

class lifo :  buf()
    method get()
        return pop(m_data)
    end
    method peek()
        if (size() > 0) then
            return m_data[1]
    end
    method put(val)
        push(m_data, val)
        return
    end
    method size()
        return *self.m_data
    end
initially()
    write("lifo::initially")
    self$buf.initially()
end

class fifo :  buf()
```

```
    method get()
        return ::get(m_data)
    end
    method peek()
        if ((n := size()) > 0) then
            return m_data[n]
    end
    method put(val)
        ::put(m_data, val)
        return
    end
    method size()
        return *self.m_data
    end
initially()
    write("fifo::initially")
    self$buf.initially()
end

procedure fifo_test()
    local f, n

    f := fifo()
    f.put("1st")
    f.put("2nd")
    n := f.get()
    write("fifo-test:  n:  ", n)
    write("fifo-test:  n:  ", f.get())
end

procedure lifo_test()
    local l, n

    l := lifo()
    l.put("1st")
    l.put("2nd")
    n := l.get()
    write("lifo-test:  n:  ", n)
    write("lifo-test:  n:  ", l.get())
end
```

```
procedure main(argv)
    fifo_test()
    lifo_test()
end
```

# APPENDIX B

# GENERATED ICON FOR SAMPLE UNICON PROGRAM

```
#line 0 "main.icn"


procedure buf_get(self)
    runerr(700, "method get()")

end

procedure buf_peek(self)
    runerr(700, "method peek()")

end

procedure buf_put(self)
    runerr(700, "method put()")

end

procedure buf_size(self)
    runerr(700, "method size()")

end

procedure buf_initially(self,val,cnt)

#line 8 "main.icn"
    write("buf::initially");
    self.m_data := list();
    if (\cnt) then {
        every i := 1 to cnt do
            self.put(self.m_data,val);
        }
    else if (\val) then
        self.put(self.m_data,val);
    return
end
```

```
record buf__state(__s,__m,m_data)
record buf__methods(get,peek,put,size,initially)
global buf__oprec
procedure buf(val,cnt)
local self,clone
initial {
    if /buf__oprec then bufinitialize()
    }
    self := buf__state(&null,buf__oprec)
    self.__s := self
    self.__m.initially(self,val,cnt) | fail
    return self
end

procedure bufinitialize()
    initial buf__oprec := buf__methods(buf_get,buf_peek,buf_put,
        buf_size,buf_initially)
end


procedure lifo_get(self)

#line 20 "main.icn"
    return pop(self.m_data);
end

procedure lifo_peek(self)

#line 23 "main.icn"
    if (self.size()>0)then
        return self.m_data[1];
end

procedure lifo_put(self,val)

#line 27 "main.icn"
    push(self.m_data,val);
    return;
end

procedure lifo_size(self)
```

```
#line 31 "main.icn"
    return *self.m_data;
end

procedure lifo_initially(self)

#line 34 "main.icn"
    write("lifo::initially");
    (self.__m.buf.  initially(self));
    return
end
record lifo__state(__s,__m,m_data)
record lifo__methods(get,peek,put,size,initially,buf)
global lifo__oprec, buf__oprec
procedure lifo()
local self,clone
initial {
    if /lifo__oprec then lifoinitialize()
    if /buf__oprec then bufinitialize()
    lifo__oprec.buf := buf__oprec
    }
    self := lifo__state(&null,lifo__oprec)
    self.__s := self
    self.__m.initially(self,) | fail
    return self
end

procedure lifoinitialize()
    initial lifo__oprec := lifo__methods(lifo_get,lifo_peek,lifo_put,
        lifo_size,lifo_initially)
end


procedure fifo_get(self)

#line 40 "main.icn"
    return get(self.m_data);
end

procedure fifo_peek(self)
```

```
#line 43 "main.icn"
    if ((n := self.size())>0)then
        return self.m_data[n];
end

procedure fifo_put(self,val)

#line 47 "main.icn"
    put(self.m_data,val);
    return;
end

procedure fifo_size(self)

#line 51 "main.icn"
    return *self.m_data;
end

procedure fifo_initially(self)

#line 54 "main.icn"
    write("fifo::initially");
    (self.__m.buf.  initially(self));
    return
end
record fifo__state(__s,__m,m_data)
record fifo__methods(get,peek,put,size,initially,buf)
global fifo__oprec, buf__oprec
procedure fifo()
local self,clone
initial {
    if /fifo__oprec then fifoinitialize()
    if /buf__oprec then bufinitialize()
    fifo__oprec.buf := buf__oprec
    }
    self := fifo__state(&null,fifo__oprec)
    self.__s := self
    self.__m.initially(self,) | fail
    return self
end
```

```
procedure fifoinitialize()
    initial fifo__oprec := fifo__methods(fifo_get,fifo_peek,fifo_put,
        fifo_size,fifo_initially)
end

#line 58 "main.icn"
procedure fifo_test();
    local f, n;

    f := fifo();
    (f.__m.put(f,"1st"));
    (f.__m.put(f,"2nd"));
    n := (f.__m.get(f));
    write("fifo-test:  n:  ", n);
    write("fifo-test:  n:  ", (f.__m.get(f)));
end

procedure lifo_test();
    local l, n;

    l := lifo();
    (l.__m.put(l,"1st"));
    (l.__m.put(l,"2nd"));
    n := (l.__m.get(l));
    write("lifo-test:  n:  ", n);
    write("lifo-test:  n:  ", (l.__m.get(l)));
end

procedure main(argv );
    fifo_test();
    lifo_test();
end
```

# APPENDIX C

# GENERATED ICON FOR SAMPLE UNICON PROGRAM UNDER PROPOSED OPTIMIZATION

```
#line 0 "main.icn"


procedure buf_get(self)
    runerr(700, "method get()")

end

procedure buf_peek(self)
    runerr(700, "method peek()")

end

procedure buf_put(self)
    runerr(700, "method put()")

end

procedure buf_size(self)
    runerr(700, "method size()")

end

procedure buf_initially(self,val,cnt)

#line 8 "main.icn"
    write("buf::initially");
    self.m_data := list();
    if (\cnt) then {
        every i := 1 to cnt do
            self.put(self.m_data,val);
        }
    else if (\val) then
        self.put(self.m_data,val);
    return
end
record buf__methods(get,peek,put,size,initially)
record buf__mdw_inst_mdw(m_data,get,peek,put,size,initially)
global buf__oprec
procedure buf(val,cnt)
local self,clone
initial {
    if /buf__oprec then bufinitialize()
    }
    self := buf__mdw_inst_mdw(,buf_get,buf_peek,buf_put,buf_size,
        buf_initially)
    self.initially(self,val,cnt) | fail
    return self
end

procedure bufinitialize()
    initial buf__oprec := buf__methods(buf_get,buf_peek,buf_put,
        buf_size,buf_initially)
end


procedure lifo_get(self)

#line 20 "main.icn"
    return pop(self.m_data);
end

procedure lifo_peek(self)

#line 23 "main.icn"
    if (self.size()>0)then
    return self.m_data[1];
end

procedure lifo_put(self,val)

#line 27 "main.icn"
    push(self.m_data,val);
    return;
end
```

```
procedure lifo_size(self)

#line 31 "main.icn"
    return *self.m_data;
end

procedure lifo_initially(self)

#line 34 "main.icn"
    write("lifo::initially");
    (self) & (buf__oprec.  initially(self));
    return
end
record lifo__methods(get,peek,put,size,initially,buf)
record lifo__mdw_inst_mdw(m_data,get,peek,put,size,initially,buf)
global lifo__oprec, buf__oprec
procedure lifo()
local self,clone
initial {
    if /lifo__oprec then lifoinitialize()
    if /buf__oprec then bufinitialize()
    lifo__oprec.buf := buf__oprec
    }
    self := lifo__mdw_inst_mdw(,lifo_get,lifo_peek,lifo_put,
        lifo_size,lifo_initially)
    self.buf := buf__oprec
    self.initially(self,) | fail
    return self
end

procedure lifoinitialize()
    initial lifo__oprec := lifo__methods(lifo_get,lifo_peek,
        lifo_put,lifo_size,lifo_initially)
end


procedure fifo_get(self)

#line 40 "main.icn"
    return get(self.m_data);
```

```
end

procedure fifo_peek(self)

#line 43 "main.icn"
    if ((n := self.size())>0)then
        return self.m_data[n];
end

procedure fifo_put(self,val)

#line 47 "main.icn"
    put(self.m_data,val);
    return;
end

procedure fifo_size(self)

#line 51 "main.icn"
    return *self.m_data;
end

procedure fifo_initially(self)

#line 54 "main.icn"
    write("fifo::initially");
    (self) & (buf__oprec.  initially(self));
    return
end
record fifo__methods(get,peek,put,size,initially,buf)
record fifo__mdw_inst_mdw(m_data,get,peek,put,size,initially,buf)
global fifo__oprec, buf__oprec
procedure fifo()
local self,clone
initial {
    if /fifo__oprec then fifoinitialize()
    if /buf__oprec then bufinitialize()
    fifo__oprec.buf := buf__oprec
    }
    self := fifo__mdw_inst_mdw(,fifo_get,fifo_peek,fifo_put,
        fifo_size,fifo_initially)
    self.buf := buf__oprec
    self.initially(self,) | fail
    return self
end

procedure fifoinitialize()
    initial fifo__oprec := fifo__methods(fifo_get,fifo_peek,
        fifo_put,fifo_size,fifo_initially)
end

#line 58 "main.icn"
procedure fifo_test();
    local f, n;

    f := fifo();
    f.put("1st");
    f.put("2nd");
    n := f.get();
    write("fifo-test:  n:  ", n);
    write("fifo-test:  n:  ", f.get());
end

procedure lifo_test();
    local l, n;

    l := lifo();
    l.put("1st");
    l.put("2nd");
    n := l.get();
    write("lifo-test:  n:  ", n);
    write("lifo-test:  n:  ", l.get());
end

procedure main(argv );
    fifo_test();
    lifo_test();
end
```

# APPENDIX D

# SAMPLE OUTPUT-INTENSIVE ICON PROGRAM

```
record rc(fname, tbl, cmnt, delim)

procedure rc_ctor(file_name, comment_char, topic_delimiter)
    local rslt;

    rslt := rc(file_name, table(), (\comment_char | "#"),
        (\topic_delimiter | ":"));
    return rslt;
end

procedure rc_iscomment(x, s)
    if (match(x.cmnt, s) | (s == "")) then
        return;
end

procedure rc_numlines(x)
    local i;

    if (/x.tbl | (*x.tbl = 0)) then
        return 0;
    every i := 1 to *x.tbl do {
        if /x.tbl[i] then
            break;
        }
    return i - 1;
end

procedure rc_read(x)
    local n, f, i, s;

    x.tbl := table();
    f := open(\x.fname, "r") | fail;
    i := 0;
    while (s := read(f)) do {
        x.tbl[i +:= 1] := s;
        if (rc_iscomment(x, s)) then
```

```
            next;
        n := find(x.delim, s);
        x.tbl[trim(s[1:\n])] := rc_trimlead(s[\n+*x.delim:0]);
        }
    close(f);
end

procedure rc_trimlead(s)
    return reverse(trim(reverse(s)));
end

procedure rc_value_get(x, topic)
    local value;

    value := \x.tbl[topic] | fail;
    return value;
end

procedure rc_value_set(x, topic, value)
    if /(x.tbl[topic]) then
        x.tbl[rc_numlines(x)+1] := string(topic) || x.delim ||
            string(value);
    x.tbl[topic] := string(value);
end

procedure rc_write(x)
    local f, i, n, s, s2;

    f := open(\x.fname, "w") | fail;
    every i := 1 to rc_numlines(x) do {
        s := \x.tbl[i];
        if (rc_iscomment(x, s)) then {
            write(f, s);
            next;
            }
        n := find(x.delim, s);
        s2 := trim(s[1:\n]);
        write(f, \s2 || x.delim || \x.tbl[\s2]) | write(f, s);
        }
    close(f);
    return;
```

```
end

procedure main(argv)
    local stem, nfiles, ntopics;

    nfiles := (integer(\argv[1]) | 32);
    ntopics := (integer(\argv[2]) | 32);
    stem := (\argv[3] | ".rctmp");
    test_rc_create(stem, nfiles, ntopics);
end

procedure test_rc_create(stem, nfiles, ntopics)
    local i, x, fname;

    \stem | fail;
    while (nfiles > 0) do {
        fname := stem || "-" || string(nfiles);
        x := rc_ctor(fname);
        every i := 1 to ntopics do
            rc_value_set(x, "topic-" || string(i), "value-" ||
                string(i));
        rc_write(x);
        nfiles -:= 1;
        }
end
```

# APPENDIX E

# SAMPLE OUTPUT-INTENSIVE UNICON PROGRAM

```
class rcfile(fname,tbl,cmnt,delim)
    method filename()
        return fname;
    end
    method iscomment(s)
        if (match(cmnt, s) | (s == "")) then
            return;
    end
    method numlines()
        local i;
        if (/tbl | (*tbl = 0)) then
            return 0;
        every i := 1 to *tbl do {
            if /tbl[i] then
                break;
            }
        return i - 1;
    end
    method read()
        local n, f, i, s;
        tbl := table();
        f := open(\fname, "r") | fail;
        i := 0;
        while (s := ::read(f)) do {
            tbl[i +:= 1] := s;
            if (iscomment(s)) then
                next;
            n := find(delim, s);
            tbl[trim(s[1:\n])] := trimlead(s[\n+*delim:0]);
            }
        close(f);
    end
    method trimlead(s)
        return reverse(trim(reverse(s)));
    end
    method valueget(topic)
```

```
        local value;
        value := \tbl[topic] | fail;
        return value;
    end
    method valueset(topic, value)
        if /(tbl[topic]) then
            tbl[numlines()+1] := string(topic) || delim ||
                string(value);
        tbl[topic] := string(value);
    end
    method version()
        return verstr;
    end
    method write()
        local f, i, n, s, s2;
        f := open(\fname, "w") | fail;
        every i := 1 to numlines() do {
            s := \tbl[i];
            if (iscomment(s)) then {
                ::write(f, s);
                next;
                }
            n := find(delim, s);
            s2 := trim(s[1:\n]);
            ::write(f, \s2 || delim || \tbl[\s2]) | ::write(f, s);
            }
        close(f);
        return;
    end
initially(file_name, comment_char, topic_delimiter)
    fname := file_name;
    cmnt := (\comment_char | "#");
    delim := (\topic_delimiter | ":");
    read();
end

procedure main(argv)
    local stem, nfiles, ntopics;

    nfiles := (integer(\argv[1]) | 32);
    ntopics := (integer(\argv[2]) | 32);
    stem := (\argv[3] | ".rctmp");
    test_rc_create(stem, nfiles, ntopics);
end

procedure test_rc_create(stem, nfiles, ntopics)
    local i, rc, fname;

    \stem | fail;
    while (nfiles > 0) do {
        fname := stem || "-" || string(nfiles);
        rc := rcfile(fname);
        every i := 1 to ntopics do
            rc.valueset("topic-" || string(i), "value-" ||
                string(i));
        rc.write();
        nfiles -:= 1;
        }
end
```

## REFERENCES

[Burg-66] Burge, William H. *A Reprogramming Machine.* Communications of the ACM, Vol. 9, Issue 2, ACM Press, Feb. 1966.

[Gris-71] Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language.* 2nd ed. Prentice-Hall, Englewood Cliffs, NJ., 1971.

[Gris-78] Griswold, Ralph E. *A History of the SNOBOL Programming Languages.* ACM Press, New York, NY., 1978

[Gris-79] Griswold, Ralph E., David R. Hanson, and John T. Korb. *The Icon Programming Language: An Overview.* ACM SIGPLAN Notices, Vol. 14, No. 4, April 1979.

[Gris-96] Griswold, Ralph E. and Madge T. *The Icon Programming Language.* 3rd Edition, Peer-To-Peer Communications, Inc., Menlo Park, CA., 1996.

[Jeff] Jeffery, Clinton L., Shamim Mohamed, Ray Pereda, and Robert Parlett. *Programming With Unicon.* 2004.

[O'Ba] O'Bagy, J. E., Ralph E. Griswold, *A Recursive Interpreter for the Icon Programming Language.* ACM SIGPLAN Notices, Papers of the Symposium on Interpreters and Interpretive Techniques, SIGPLAN '87, Vol. 22, Issue 27. July, 1987.

[Walk-91] Walker, Kenneth W. *An Optimizing Compiler for Icon.* University of Arizona CS TR 91-16.