The Revised Revised Revised Revised Revised Report on CCL: A Subject Language for Compiler Projects

Alan Zaring
Luther College
Computer Science Dept.
Decorah, IA 52101
USA
email: akzaring@luther.edu
phone: 563-387-1137

ABSTRACT

This paper presents CCL, a programming language designed for implementation efforts and laboratory projects in compilers courses for undergraduates. CCL is small enough, cohesive enough, and regular enough that a working compiler can be implemented by an individual student in a single semester; yet, CCL includes a collection of features (including pointers, dynamically-sized arrays, anonymous functions, and a restricted form of continuations) that make it a useful vehicle for studying compiler theory and design. Experiences using CCL in compilers courses are discussed, and suggestions for a number of projects concerning CCL are given.

INTRODUCTION

CCL (Zaring, 2001) is a language specifically designed for implementation projects in a compiler theory and design course for undergraduates. CCL is small enough that an individual student can implement a complete CCL compiler in a single semester. (If desired, judicious subsetting can reduce the size of the language even further.) Yet, CCL also incorporates a number of advanced programming language concepts and can provide students with more insights into programming language design, semantics, and compilation than might be gained from implementing a compiler for stripped-down versions of larger, more complex programming languages.

The constructs and concepts in CCL are based on a number of programming languages, including Algol 68 (van Wijngaarden et al., 1975), C (Kernighan & Ritchie, 1988), C++ (Stroustrup, 2013), Pascal (Jensen et al., 1992), Scheme (Sperber at al., 2007), and, to a great degree, Toy (Demers & Teitelbaum, 1980). CCL is a strongly-typed, statically-typed, lexically-scoped language. It is expression-oriented and allows functions to be used quite generally. It includes a number of features that are valuable and challenging for students to master and implement: pointers, dynamically-sized arrays, and continuations, among others.

For brevity and economy, a number of common features are missing from CCL. For example, CCL has no character type, since it was felt that implementing such a type added relatively little additional insight into the compiler-design process. More notably, CCL includes no object-oriented features, although some object-oriented features can be added with relative ease.

ANNOTATED GRAMMAR FOR CCL

We develop CCL by presenting a BNF grammar for the language. Each group of related productions is accompanied by comments explaining the semantics and pragmatics of that portion of the language. The complete grammar appears in Appendix A and is also available, in various formats, from the author. As presented, the grammar is designed for use by bottom-up parsers, like those generated by Yacc (Johnson, 1975) or Bison (Donnelly & Stallman, 2002).

The semantics of CCL are reasonably straightforward and are consistent with a stack-based implementation of a run-time environment. The semantics of function application are consistent with lexical-scoping conventions.

CCL obeys typical lexical conventions:

- Adjacent keywords and/or identifiers, unless separated by punctuation marks, must be separated by at least one *whitespace* character (i.e., a space, a tab, or a newline).
- Whitespace cannot appear in the midst of a *token* (i.e., a single atomic syntactic entity) of the language.
- An exclamation point (“`!`”) begins a “to the end of the line” comment. All characters from the exclamation point up to but not including the next newline are ignored.

Functions

The CCL function mechanism is similar to the procedure mechanism found in Scheme (Sperber at al., 2007), except that, in CCL, formal parameters and return values are strictly typed, and functions are not first-class values.

(1) *program*
(1a) ::= *functionConstant*

A CCL program is a function having no formal parameters. A CCL program is executed by implicitly applying this function. The value of this application is ignored and discarded.

(2) *functionConstant*
(2a) ::= `fun` `(` *formalParameterDeclarationPart* `)` *expressionSequence*
`endfun`

(3) *formalParameterDeclarationPart*
(3a) ::= ε
(3b) | *formalParameterDeclarationList*

(4) *formalParameterDeclarationList*
(4a) ::= *formalParameterDeclaration*
(4b) | *formalParameterDeclarationList* `,` *formalParameterDeclaration*

Function constants denote unnamed values (as do Scheme lambda-expressions), in contrast with Pascal, C, and C++ function definitions (which do not denote values and are obligatorily named). The value of a function constant is, in itself, useless. However, the value of a function constant can be applied to actual parameters (see production (14h)), with the value resulting from such an application and any side-effects occurring during that application likely being of interest. Function constants are deemed “constants” since simply evaluating a CCL function constant does not cause observable work to be performed (although the application of the value of that function constant at some later time may cause observable work to be done).

Functions have zero or more formal parameters. There can be at most one declaration for any given identifier in a given formal parameter declaration list. Each formal parameter is accessible anywhere in the expression sequence comprising the body of the function except where hidden by a nested declaration of a formal parameter or local variable (see productions (19)-(23)) of the same name.

All parameters in CCL are passed by value (with the phrase "by value" used here in the same sense as in Pascal, C, and C++). Call-by-reference can be simulated through the use of explicit references (see productions (13q) and (17c)), while call-by-name can be simulated through the use of functions as parameters. When a function is applied, the value resulting from evaluating the function body is returned as the value of the application; the type of value returned by applications of the function is inferred from the type of the function's body. Function constants can be nested arbitrarily.

The run-time *extent* (i.e., lifetime) of the value of a function constant *F* is limited to a single execution of the body of the function constant or variable block (see production (19a)) that most closely textually encloses *F*. Under this interpretation, function values can be passed as actual parameters with full generality, but they cannot be returned as values of function applications with full generality. Thus, CCL supports *downward funargs* but does not support *upward funargs* (Friedman et al., 2008; Moses, 1970).

(5) *formalParameterDeclaration*
(5a) ::= *formalParameterType identifier*

Formal parameter declarations are reminiscent of those in C and C++, with the type appearing first followed by the formal parameter's name. Note, however, that there can be only one identifier per declaration.

Formal Parameter Types

Formal parameters of CCL functions can be declared to be either *mutable* (meaning that they can be assigned to) or *immutable* (meaning that they cannot be assigned to). By default, formal parameters are mutable.

(6) *formalParameterType*
(6a) ::= *unqualifiedFormalParameterType*
(6b) | `immut` *unqualifiedFormalParameterType*

Formal parameters of CCL functions can be declared to hold values of any of six basic kinds: meaningless values (similar in spirit to type `void` in C), integers, pointers, vectors, continuations, or functions.

(7) *unqualifiedFormalParameterType*
(7a) ::= `triv`

`triv` is a type containing a single unspecified value and is typically used as the return type for functions designed to achieve results through side-effects (as is done by Pascal procedures and by C and C++ functions having return type `void`).

(7b) | `int`

`int` is the type of signed integers. The range of integer values in type `int` is implementation-dependent.

The remaining types permit the construction of more complex types from simpler types.

(7c) | `ref` *formalParameterType*

`ref` values are type-restricted pointers, similar to the pointer types in Pascal, C, and C++. References point to formal parameters or to local variables (see productions (19)-(23)), not to values. A given reference value may refer to a local variable or formal parameter that has gone out of extent; thus, dangling reference phenomena may occur during program execution.

(7d) | `ref vec` *formalParameterType*

A CCL homogenous, ordered aggregate is called a *vector*. Vectors are a kind of run-time entity in CCL, but they are not a kind of run-time value: there are no vector-valued expressions. Vectors are manipulated via references to vectors (also called *ref-vecs*). All vector operations require ref-vec values. Vector declarations (see production (23a)) declare a ref-vec local variable, not a vector local variable. Like C and C++, but unlike Pascal, CCL uses an "indirect" representation of arrays (Friedman et al., 2008). As with any CCL reference type, dangling-reference phenomena may occur.

(7e) | `con` *formalParameterType*

CCL includes a downward-continuation mechanism for short-circuiting the expression evaluation process. This mechanism is similar to the escape-continuation mechanism found in some versions of Scheme (e.g., Flatt, 2016), to the mechanism provided by the functions `setjmp` and `longjmp` in the standard C library (Plauger, 1992), to the exception-handling mechanism provided by try-blocks and `throw` expressions in C++ (Stroustrup, 2013), and to the mechanism provided by the `catch` and `throw` forms in Common LISP (Steele, 1990). CCL's method makes use of the *control block* expression (see production (14f)), the `continues` operator (see production (11b)), and `con` ("continuation") values. Continuation values, formal parameters, and local variables are typed to indicate what type of value they must yield upon use.

(7f) | `fun (` *formalParameterTypePart* `)` *formalParameterType*

(8) *formalParameterTypePart*
(8a) ::= ε
(8b) | *formalParameterTypeList*

(9) *formalParameterTypeList*
(9a) ::= *formalParameterType*
(9b) | *formalParameterTypeList* `,` *formalParameterType*

Expressions, formal parameters, and local variables can be of function types. The specification of a function type includes the types of the function's formal parameters and the function's return value. In such specifications, only the types of the formal parameters are specified, not their respective names.

Expressions

CCL is an expression-oriented language, with a fairly rich collection of operators and expressions, both for producing values and for controlling the order of execution within a program.

(10) *expressionSequence*
(10a) ::= *expression*
(10b) | *expressionSequence* `;` *expression*

In many places in CCL where an expression may occur, a semicolon-separated sequence of expressions may appear. At run time, the expressions are evaluated sequentially from first to last, with the values of all but the last expression discarded. The value of the last expression is returned as the value of the sequence. The type of the sequence is determined by the type of the last expression.

The remainder of the language description deals with the relatively large number of expressions and operators in the language. CCL has no statements in the sense that languages such as Pascal, C, and C++ have: all expressions in CCL produce a value. The precedences and associativities for the expressions and operators in the language are

expressions/operators	*precedence*	*associativity*
`()`'s for grouping	7	none
`while`, `if`, `control`, `vars`	7	none
`()`'s for function applications, `[]`, `@`	7	left
`input`, `output`, unary +, unary −, `#`, `&`	6	right
`*`, `/`	5	left
`+`, `-`	4	left
=, <>, <=, <, >=, >	3	left
`:=`	2	right
`continues`	1	right

Unless otherwise specified, the operands of an operator are evaluated from left to right, sequentially.

(11) *expression*
(11a) ::= *assignmentExpression*
(11b) | *assignmentExpression* `continues` *expression*

The `continues` operator is used in conjunction with `con` values and control blocks to implement CCL's downward-continuation mechanism. The first operand must be an expression of a continuation type. If the first operand is of type `con` *T* or `immut con` *T*, then the second operand must be an expression of type *U*, where *U* is a *subtype* of *T* (see the section below on the formal properties of CCL types). The value of the operation is the value of the second operand, converted to have type *T* (but see production (14f) for a detailed discussion).

(12) *assignmentExpression*
(12a) ::= *simpleExpression*
(12b) | *variableExpression* `:=` *assignmentExpression*

The assignment operator `:=` assigns the value of the right-hand side expression to the left-hand side variable. (The term "variable" in CCL includes both formal parameter and local variables.) The inferred type of the left-hand side variable (see production (17)) must be a supertype of the type of the right-hand side expression. The value of the operation is the newly-assigned value of the left-hand side variable.

Binary and Unary Operators

The following group of productions describes a typical collection of binary and unary operators for comparison, arithmetic, etc. For the sake of brevity, these productions are ambiguous and do not fully capture the precedences and associativities of the operators.

(13) *simpleExpression*
(13a) ::= *primaryExpression*
(13b) | *simpleExpression* `=` *simpleExpression*
(13c) | *simpleExpression* `<>` *simpleExpression*
(13d) | *simpleExpression* `<=` *simpleExpression*
(13e) | *simpleExpression* `<` *simpleExpression*
(13f) | *simpleExpression* `>=` *simpleExpression*
(13g) | *simpleExpression* `>` *simpleExpression*

The comparison operators return the integer value zero for "false" and the value one for "true." The first and second operands must be of "compatible" types (see the section below on the formal properties of CCL types). Values of subtypes of types `int` and `triv` can be compared using any of the six comparisons. Values of subtypes of types `ref`, `ref vec`, and `con` can be compared for equality and inequality only. Values of subtypes of type `fun` cannot be compared at all (the rationale for this stemming from a lack of general agreement concerning the equality of unnamed function values (Sperber at al., 2007)).

(13h) | *simpleExpression* `+` *simpleExpression*
(13i) | *simpleExpression* `-` *simpleExpression*

The addition and subtraction operators can be applied to values of subtypes of type `int` only. The result value is of type `int`. Typical CCL compilers generate arithmetic expression code that makes no special provisions for arithmetic overflow.

(13j) | *simpleExpression* `*` *simpleExpression*
(13k) | *simpleExpression* `/` *simpleExpression*

The multiplication and division operators can be applied to values of subtypes of type `int` only. The result value is of type `int`. The division operator returns only the quotient, discarding the remainder.

(13l) | `input` *variableExpression*

The `input` operator evaluates its operand to a variable of type `int`, reads a single integer from the standard input source, and assigns that value to its variable operand. The `int` value read in is returned as the value of the input expression. A typical implementation of CCL uses whatever notion of “standard input source” is customary for the system on which it is implemented.

(13m) | `output` *simpleExpression*

The `output` operator writes the value of its operand (which must be of a subtype of type `int`) to the standard output sink. The value written out, converted to type `int`, is returned as the value of the output expression. A typical implementation of CCL would use whatever notion of “standard output sink” is customary for the system on which it is implemented and would terminate the current output line after writing out the integer value.

(13n) | `+` *simpleExpression*
(13o) | `-` *simpleExpression*

The unary plus and minus operators can be applied to values of subtypes of type `int` only. The result value is of type `int`. The unary plus operator performs no observable function. The unary minus operator negates its operand.

(13p) | `#` *simpleExpression*

The *vector-length operator* can be applied to values type `ref vec` or `immut ref vec` only (i.e., values of subtypes of type `ref vec`). It returns the number of elements (as a value of type `int`) in the vector referred to.

(13q) | `&` *variableExpression*

The *reference operator* can be applied to variables (i.e., local variables and formal parameters) only. It returns a reference (a pointer) to its operand. If the operand is of type *T*, the reference value is of type `ref` *T*.

Control Structures

The next group of productions describes expressions that provide the sorts of control structures normally provided by “statements” in imperative programming languages.

(14) *primaryExpression*
(14a) ::= *constant*
(14b) | *variableExpression*

In this context, the value of a variable expression is the value contained in that variable at that moment.

(14c) | `(` *expressionSequence* `)`

Grouping-parentheses are used primarily to cause lower-precedence operations to be done before higher-precedence operations, but they also permit sequences of expressions to be used in contexts where a single, simple expression is required.

(14d) | `while` *expressionSequence* `do` *expressionSequence* `endwhile`

The `while` expression provides an iterative-control mechanism. At run time, the first operand (the *termination test*), which must be of a subtype of type `int`, is evaluated. If the termination test has value zero, the loop stops, and the `int` value zero is returned as the value of the loop; if, instead, the termination test has a non-zero value, the second operand (the *body*) is evaluated and its value discarded. This sequence of events is then repeated, starting with the re-evaluation of the termination test. A `while` expression will either fail to terminate or will terminate and return the value zero; thus, the termination test or the body must perform side-effects if the loop is to perform observable work.

(14e) | `if` *expressionSequence* `then` *expressionSequence* `else`
expressionSequence `endif`

The `if` expression provides a conditional-control mechanism. At run time, the first operand (the *selector*), which must be of a subtype of type `int`, is evaluated. If the selector has value zero, the third operand (the *false clause*) is evaluated and that value returned as the value of the `if` expression. If the selector instead has a non-zero value, the second operand (the *true clause*) is evaluated and that value returned as the value of the `if` expression. The false clause is mandatory (as is common in expression-oriented languages). The values of the true and false clauses must be of "compatible" types (see the section below on the formal properties of CCL types).

(14f) | `control` *variableExpression* `in` *expressionSequence*
`endcontrol`

The `control` expression (or *control block*) is part of the downward-continuation (or "escape-continuation") mechanism for short-circuiting expression evaluation. The first operand (the *control variable*) must be a variable of type `con` *T* (for any type *T*). The second operand (the *body*) must then be of a subtype of type *T*.

Consider the run-time evaluation of the expression

`control` *controlVar* `in` *body* `endcontrol`

First, *controlVar* is assigned the "continuation value" for this evaluation of this control block; call this continuation value *C*. Then, *body* is evaluated. Normally, the value of *body* is returned as the value of the control block.

However, suppose that at some time during this execution of *body* an expression of the form

conExpr `continues` *valueExpr*

(see production (11b)) is evaluated. First *conExpr* is evaluated and returns a continuation value; suppose this value equals C. Then, *valueExpr* is evaluated; call this value V. Finally, the value V is immediately (with all remaining and pending uncompleted computations from *body* discarded) returned as the value of the control block whose continuation value was C. Note that the `continues` expression need not be textually included in the body of the control block: it might, for example, be present in the body of a function applied from within the body of the control block.

The extent of the continuation value produced upon entering a control block is limited to that single execution of that control block. It is possible for "dangling-continuation" phenomena to occur. Control blocks can be nested. Two continuation values are considered equal only if they were initially created by the same execution of the same control block.

(14g) | *variableBlock*

A variable block expression provides a mechanism by which local variables can be created and used. See production (19a) for details.

(14h) | *primaryExpression* `(` *actualParameterPart* `)`

(15) *actualParameterPart*
(15a) ::= ε
(15b) | *actualParameterList*

(16) *actualParameterList*
(16a) ::= *expressionSequence*
(16b) | *actualParameterList* `,` *expressionSequence*

In function application expressions, the first operand must be an expression of a `fun` type, say `fun(`T_1`, ... ,` T_n`)` T_{n+1}. The actual parameter list must then be a comma-separated list of n expression sequences, with the j^{th} sequence being of a subtype of type T_j ($1 \leq j \leq n$). At run time, the first operand is evaluated, the actual parameters are evaluated (in an unspecified order), the function is applied to the actual parameter values, and the function returns a value of type T_{n+1} to the point of application.

Variables

The next group of productions specifies the constructs that can appear in contexts where a variable is required (e.g., on the left-hand side of an assignment expression).

(17) *variableExpression*
(17a) ::= *identifier*

An identifier serves as the name of a variable. All variables must be declared before they can be used. Both local variables declared in variable blocks (see production (19a)) and formal parameters of functions are considered variables.

(17b) | *primaryExpression* `[` *expressionSequence* `]`

The *index operation* is used to access individual elements of vectors. The first operand must be of a subtype of type `ref vec` *T*. The second operand (the *subscript*) must be of a subtype of type `int`. The subscript must evaluate to a non-negative value less than the number of elements in the vector referred to by the first operand. If the subscript has value *J*, the index expression stands for element *J* of the vector referenced by the first operand (with the first element in a vector always having subscript zero).

(17c) | *primaryExpression* `@`

CCL's *dereference operator* is similar to the `^` operator in Pascal and the unary `*` operator in C and C++. Its operand must be an expression of a subtype of type `ref` *T*, but cannot be an expression of a subtype of type `ref vec` *T*. The expression denotes the variable referred to (i.e., pointed to) by the reference value.

Constants

CCL has only a relatively small number of constructs for representing literal values.

(18) *constant*
(18a) ::= *integerConstant*
(18b) | `?`
(18c) | *functionConstant*

Integer constants denote values of type `int` and are discussed in production (25a). `?` denotes the one and only value of type `triv`. Function constants were discussed previously.

Local Variables

Local variables can be introduced into a CCL program at any point through the use of variable blocks.

(19) *variableBlock*
(19a) ::= `vars` *variableDeclarationList* `in` *expressionSequence* `endvars`

(20) *variableDeclarationList*
(20a) ::= *variableDeclaration*
(20b) | *variableDeclarationList* `,` *variableDeclaration*

The `vars` expression (or *variable block*) is similar to the `let*` form in Scheme and the compound statement in C and C++, providing a means by which local variables can be declared and created. Each local variable declared in a variable block's declaration list is accessible anywhere in the remainder of that declaration list as well as in the body of the block, except where hidden by the declaration of a nested formal parameter or local variable of the same name. There can be at most one declaration for any given identifier in a given variable declaration list. Variable blocks can be nested arbitrarily.

Variables local to a block come into being (in the same order in which their respective declarations appear textually in the block) just prior to each execution of the block body and then cease to exist each time the execution of the block body terminates. The initial value of a newly-created local variable is undefined, unless that local variable was declared as a vector. Local variables do not retain values across multiple executions of the block. The value of a variable block is the value of its body.

Local Variable Types and Vectors

Local variables can be either mutable or immutable and can hold the same types of values as can formal parameters of functions. In addition, vectors can be introduced through the use of local variables.

(21) *variableDeclaration*
(21a) ::= *variableType* *identifier*

(22) *variableType*
(22a) ::= *formalParameterType*
(22b) | *unqualifiedVariableType*
(22c) | `immut` *unqualifiedVariableType*

(23) *unqualifiedVariableType*
(23a) ::= `vec` `[` *expressionSequence* `]` *variableType*

Local variables declared in a variable block can be of the same types as formal parameters of functions or can be of one additional type: a local variable can be declared to be a *vector*. The *length* of a vector (i.e., the number of elements in the vector) is specified by an expression sequence of a subtype of type `int`, which must evaluate to a non-negative value. The length expression is evaluated anew each time the local variable is created; thus, CCL vectors have dynamic length. The length of a vector is considered to be part of the value of the vector and is not considered to be part of the type of the vector. The extent of the vector is limited to the remainder of the execution of the local-variable declaration list (if any) and the subsequent execution of the body of the variable block. Vectors whose elements are themselves vectors ("vectors of vectors" or so-called "multi-dimensional vectors") are permitted, although basic implementations of CCL may choose to permit only "one-dimensional" vectors.

Since there are no expressions having values of type `vec` *T* (as was mentioned in the discussion following production (7d)), the declaration of `V` in

```
vars vec [17] int V in … endvars
```

declares `V` to be a local variable of type `ref vec int` (and not a variable of type `vec int`). When the declaration for `V` is executed, a vector of 17 integers is created, and `V` is initialized to a reference to that vector (where initialization is similar to, yet not identical to, assignment, as is the case in C++ (Stroustrup, 2013)). All vector-related operators (the index operator and the length operator) require operands of type `ref vec` *T*. A ref-vec local variable declared in a vector declaration is said to be *vector-declared*.

The situation generalizes in the case of a vector of vectors. The length expressions are evaluated sequentially from outermost to innermost, with each expression being evaluated once (and only once) each time the variable declaration is executed, and the ref-vec's are initialized from outermost to innermost. If `V` were declared as

```
vars vec [17] vec [18] int V in … endvars
```

a vector of 17 ref-vec's would be created, and `V` would be initialized to a reference to that vector. Then, 17 vectors, each having 18 `int` elements, would be created, and each of `V[0]` through `V[16]` would be initialized to point to its own individual vector.

Immutable local variables are generally of little use, since they can never be assigned meaningful values. However, immutable vectors can be useful. In the preceding example, the vector-declared ref-vec local variable `V` could be assigned a new value (e.g., a reference to some other vector of vectors of integers), perhaps losing the only reference to the vector `V` initially pointed to. To prevent `V` from being made to point to a vector other than the one it initially pointed to, `V` can instead be declared to be immutable:

```
vars immut vec [17] vec [18] int V in … endvars
```

While this declaration makes `V` immutable, the elements of the vector that `V` references are mutable. To prevent the elements of `V` from being made to point to other vectors, `V` would be declared as

```
vars vec [17] immut vec [18] int V in … endvars
```

To prevent all the ref-vec's associated with the declaration of `V` from being reassigned, `V` would be declared as

```
vars immut vec [17] immut vec [18] int V in … endvars
```

Note that the length expression-sequence in a vector declaration is evaluated in a data environment that does not include the ref-vec local variable being declared. So, in the variable block

`vars` t_1 i_1, … , t_{j-1} i_{j-1}, `vec` [$expr_j$] t_j i_j, t_{j+1} i_{j+1}, … , t_n i_n `in` … `endvars`

$expr_j$ is evaluated in an environment which does contain the definitions for this block's local variables i_1, ... , i_{j-1} but does not contain definitions for this block's local variables i_j , ... , i_n.

Identifiers and Literals

Every implementation of CCL must provide a specification of the permitted forms for identifiers and for literals of type `int`.

(24) *identifier*
(24a) ::= ...

In typical implementations, identifiers consist of sequences of one or more letters, digits, and underscore characters. Identifiers cannot start with a digit.

(25) *integerConstant*
(25a) ::= ...

In typical implementations, integer constants consist of sequences of one or more decimal digits. Again, the sizes of the largest positive and smallest negative values permitted are implementation-dependent.

FORMAL PROPERTIES OF CCL TYPES

If immutable types were absent from CCL, type-checking would be nearly trivial since none of the basic types (`triv`, `int`, `ref`, `ref vec`, `con`, and `fun`) are related or interchangeable in any meaningful way. Without immutable types, in the above discussions of the type-related aspects of CCL, the phrase "type *T* is a subtype of type *U*" could simply be replaced by the phrase "type *T* is equal to type *U*."

The presence of immutable types complicates matters. For example, it is quite reasonable to assume that there is a close relationship and high degree of interchangeability between values of type `int` and values of type `immut int`. It is also reasonable to assume there is some relationship between the types `ref int`, `ref immut int`, `immut ref int`, and `immut ref immut int`, although the exact nature of the relationship and the degree of interchangeability is more complex.

The sections that follow provide formal definitions of relations for "type *T* is a subtype of type *U*" and "expression *x* is of type *T*". The relations are defined as collections of inference rules, with each rule having the form

$$\frac{premise_1 \quad \ldots \quad premise_n}{conclusion}$$

Such a rule states that if all the premises are true, the conclusion is true. If there are no premises in a rule, the conclusion is always true (i.e., the rule is an axiom).

Italicized, capital letters (e.g., "*T*") are used as *type variables* in the rules. If such a type variable appears in a premise but not in the conclusion, and its value is never used directly, that variable should be thought of as being implicitly existentially quantified.

Although we have earlier referred to the type `triv`, the type `int`, etc., it is important to avoid confusion in the rules between *type expressions* (i.e., the concrete syntax used to designate types in formal parameter and variable declarations) and the types to which they correspond. To this end, a different notation is used to denote the types:

type expression	*type*
`immut` T	$\mathit{Immut}\ T$
`triv`	Triv
`int`	Int
`ref` T	$\mathit{Ref}\ T$
`ref vec` T	$\mathit{Refvec}\ T$
`con` T	$\mathit{Con}\ T$
`fun (`T_1`, ... , `T_j`)` U	$\mathit{Fun}\ (T_1, \ldots, T_j)\ U$

The complete collections of rules appear without commentary in Appendix B.

Subtypes

We say "type T is a subtype of type U" (or, alternatively, "type U is a supertype of type T"), written as

$$T <: U$$

if values of type T may be safely used in all situations where values of type U are expected (Pierce, 2002). Based on this notion of subtyping as "safe substitution in all instances," we can define the <: relation for CCL.

The first rule states that type `triv` is a subtype of itself:

$$\frac{}{\mathit{Triv} <: \mathit{Triv}}$$

The rule may be read as "type `triv` is a subtype of type `triv`" or "a value of type `triv` can be used wherever a value of type `triv` is expected."

The second rule states the analogous property for type `int`:

$$\frac{}{\mathit{Int} <: \mathit{Int}}$$

The third and fourth rules specify the relationship between mutable and immutable types:

$$\frac{T <: U}{T <: \mathit{Immut}\ U}$$

$$\frac{T <: U}{\mathit{Immut}\ T <: U}$$

Here, "$\mathit{Immut}\ V$" is used to denote the type that is like type V but with an added outermost "`immut`".

The fifth and sixth rules specify the relationship among `ref` types:

$$\frac{T <: U \qquad \text{mutable?}(T) \qquad \text{mutable?}(U)}{\mathit{Ref}\ T <: \mathit{Ref}\ U}$$

$$\frac{T <: U \qquad \text{immutable?}(U)}{\mathit{Ref}\ T <: \mathit{Ref}\ U}$$

("$\mathit{Ref}\ V$" is used in a manner analogous to the earlier use of "$\mathit{Immut}\ V$".) Defining the expression "mutable?(V)" to be true if and only if type V is a type that is mutable at the outermost level (e.g., `ref immut int`) and defining the expression "immutable?(V)" to be true if and only if type V is immutable at the outermost level (e.g., `immut ref int`), from these rules, one can infer, for example,

`ref int` <: `ref int`

`ref int` <: `ref immut int`

$$\texttt{ref immut int} <: \texttt{ref immut int}$$

However, as intended, one cannot infer

$$\texttt{ref immut int} <: \texttt{ref int}$$

Consider the example

```
vars
    ref int ri,
    ref immut int rii
in
    …
    ri := rii;
    ri @ := 17;
    …
    endvars
```

If the assignment of `rii` to `ri` were permitted, the assignment to `ri @` would in fact modify the immutable variable pointed to by `rii`. To preclude such inappropriate modifications, it must not be the case that `ref immut` T <: `ref` T. The seventh and eighth rules provide analogous specifications for references to vectors:

$$\frac{T <: U \qquad \text{mutable?}(T) \qquad \text{mutable?}(U)}{\mathit{Refvec}\ T <: \mathit{Refvec}\ U}$$

$$\frac{T <: U \qquad \text{immutable?}(U)}{\mathit{Refvec}\ T <: \mathit{Refvec}\ U}$$

The ninth rule specifies the relationship among `con` types:

$$\frac{T <: U}{\mathit{Con}\ T <: \mathit{Con}\ U}$$

The final rule specifies the relationship among `fun` types:

$$\frac{V_1 <: T_1 \qquad V_2 <: T_2 \qquad \ldots \qquad V_j <: T_j \qquad U <: W}{\mathit{Fun}\ (T_1,\ \ldots\ ,\ T_j)\ U <: \mathit{Fun}\ (V_1,\ \ldots\ ,\ V_j)\ W}$$

Loosely paraphrasing the rule, a function F may be substituted for a function G if F can be applied to all the sorts of actual parameters that G could be applied to (and perhaps more) and if F can never produce a sort of value that G could not produce. Put another way, F must be no less liberal than G with respect to its actual parameters and no more liberal than G with respect to its return value.

For convenience only, rules expressing the reflexivity and transitivity of the <: relation are also included:

$$\frac{}{T <: T}$$

$$\frac{T <: U \qquad U <: V}{T <: V}$$

Type-Checking Rules

Having defined the subtype relation, it is possible to present a collection of inference rules to define the relation

$$x : T$$

(read as "expression x is of type T") that formalizes the type-checking rules for CCL. The rules are annotated with the number of the grammar production for the expression whose type-checking rule is being defined and assume the same conventions, definitions, and notations used in defining the subtype relation.

Note that the set of rules given here is somewhat informal and incomplete. A rigorous and complete set of type-checking rules would not only need to capture additional notions (particularly various issues surrounding identifier declaration, scope rules, and execution) in a manner similar to that demonstrated in Pierce, 2002, but would also need to address the issue of assigning types to type-erroneous expressions and programs (perhaps by assigning such constructs a designated "error" or "bottom" type). However, the rules as presented provide a sufficient basis for developing a CCL compiler.

From production (2):

$$\frac{x_1 : T_1 \quad \ldots \quad x_n : T_n \quad y : U}{\texttt{fun (}x_1\ id_1\texttt{,}\ \ldots\ x_n\ id_n\texttt{)}\ y\ \texttt{endfun} : \mathit{Fun}\ (T_1, \ldots , T_n)\ U}$$

The type of a function constant is determined by the declared types of its formal parameters together with the inferred type of its body.

From production (6):

$$\frac{x : T}{\texttt{immut}\ x : \mathit{Immut}\ T}$$

From production (7a):

$$\frac{}{\texttt{triv} : \mathit{Triv}}$$

From production (7b):

$$\frac{}{\texttt{int} : \mathit{Int}}$$

From production (7c):

$$\frac{x : T}{\texttt{ref}\ x : \mathit{Ref}\ T}$$

From production (7d):

$$\frac{x : T}{\texttt{ref vec}\; x : \mathit{Refvec}\; T}$$

From production (7e):

$$\frac{x : T}{\texttt{con}\; x : \mathit{Con}\; T}$$

From production (7f):

$$\frac{x_1 : T_1 \quad \ldots \quad x_n : T_n \quad y : U}{\texttt{fun (}x_1\texttt{,}\; \ldots\, x_n\texttt{)}\; y : \mathit{Fun}\; (T_1, \ldots, T_n)\; U}$$

From production (10):

$$\frac{x_1 : T_1 \quad x_2 : T_2 \quad \ldots \quad x_n : T_n}{x_1\; \texttt{;}\; x_2\; \texttt{;}\; \ldots\; \texttt{;}\; x_n : T_n}$$

Since the value of an expression sequence is the value of its last expression, the type of an expression sequence is given by the type of its last expression.

From production (11b):

$$\frac{x : \mathit{Con}\; T \quad y : U \quad U <: T}{x\; \texttt{continues}\; y : T}$$

$$\frac{x : \mathit{Immut}\; \mathit{Con}\; T \quad y : U \quad U <: T}{x\; \texttt{continues}\; y : T}$$

Since executing a `continues` expression causes the immediate resumption of the execution of some control block, the value of the `continues` expression itself is somewhat moot. However, the type of that value must be specified to permit the unambiguous typing of the expression enclosing a `continues` expression. One could specify a fixed, arbitrary type (e.g., `triv`) as the type of all `continues` expressions; however, making an analogy between returning from a function and resuming a control block, a type based on the types of the operands seems appropriate.

Having a pair of rules for the `continues` expression (in which the only difference is the immutability of the type of the first operand) seems awkward. One might be tempted to replace the pair with the single rule

$$\frac{x : T \quad y : U \quad T <: \mathit{Con}\; V \quad U <: V}{x\; \texttt{continues}\; y : V}$$

However, although this rule is logically correct, it is imprecise: V can be any type satisfying $T <: \mathit{Con}\; V$. Using instead the pair of rules given previously, the type of a `continues` expression can be stated precisely in terms of the types of its two operands.

From production (12b):

$$\frac{x : T \quad y : U \quad \text{mutable?}(T) \quad U <: T}{x \mathrel{:=} y : T}$$

The assignment operation is meaningful only if the left-hand side variable is both mutable and capable of receiving the value of the right-hand side expression. If it is meaningful, its type is that of the left-hand side variable.

From productions (13b)-(13g):

$$\frac{x : T \quad y : U \quad T <: \mathit{Triv} \quad U <: \mathit{Triv}}{x\ \mathit{op}\ y : \mathit{Int}}$$

$$\frac{x : T \quad y : U \quad T <: \mathit{Int} \quad U <: \mathit{Int}}{x\ \mathit{op}\ y : \mathit{Int}}$$

where *op* is any of <, <=, >, >=, =, or <>. Both `triv` and `immut triv` values as well as `int` and `immut int` values can be compared using any of the comparison operators.

$$\frac{x : T \quad y : U \quad T <: \mathit{Ref}\ V \quad \text{compatible?}(T, U)}{x\ \mathit{op}\ y : \mathit{Int}}$$

$$\frac{x : T \quad y : U \quad T <: \mathit{Refvec}\ V \quad \text{compatible?}(T, U)}{x\ \mathit{op}\ y : \mathit{Int}}$$

$$\frac{x : T \quad y : U \quad T <: \mathit{Con}\ V \quad \text{compatible?}(T, U)}{x\ \mathit{op}\ y : \mathit{Int}}$$

where *op* is = or <>, and the expression "compatible?(V, W)" is defined to be true if and only if $V <: W$ or $W <: V$. `ref`, `ref vec`, and `con` values (as well as `immut ref`, `immut ref vec`, and `immut con` values) can be compared only with the = and <> operators. Note that the variable V is implicitly existentially quantified in these rules, as explained earlier.

From productions (13h)-(13k):

$$\frac{x : T \quad y : U \quad T <: \mathit{Int} \quad U <: \mathit{Int}}{x\ \mathit{op}\ y : \mathit{Int}}$$

where *op* is any of +, −, `*`, or `/`. Both `int` and `immut int` values can be operated on using any of the binary arithmetic operators.

From production (13l):

$$\frac{x : \mathit{Int}}{\texttt{input}\ x : \mathit{Int}}$$

Only (mutable) `int` variables can have values read into them.

From production (13m):

$$\frac{x : T \qquad T <: \mathit{Int}}{\texttt{output}\ x : \mathit{Int}}$$

Both `int` and `immut int` values can be written out by the `output` operator.

From productions (13n)-(13o):

$$\frac{x : T \qquad T <: \mathit{Int}}{op\ x : \mathit{Int}}$$

where *op* is unary + or unary −. Both `int` and `immut int` values can be operated on using any of the unary arithmetic operators.

From production (13p):

$$\frac{x : T \qquad T <: \mathcal{R}\mathit{efvec}\ U}{\#\ x : \mathit{Int}}$$

From production (13q):

$$\frac{x : T}{\texttt{\&}\ x : \mathcal{R}\mathit{ef}\ T}$$

From production (14c):

$$\frac{x : T}{\texttt{(}\ x\ \texttt{)} : T}$$

From production (14d):

$$\frac{x : T \qquad y : U \qquad T <: \mathit{Int}}{\texttt{while}\ x\ \texttt{do}\ y\ \texttt{endwhile} : \mathit{Int}}$$

Note that neither the type of the termination test nor the type of the body of a `while` loop has any effect on its resultant type.

The type-checking rule corresponding to production (14e) is somewhat problematic. The type of value produced by an `if` expression must be unambiguously specified, even though the run-time value may come from either the true-clause or the false-clause. As a minimal requirement, if the type of the true-clause is *U*, and the type of the false-clause is *V*, the type of the `if` expression must be some type *W*, where *W* is a supertype of both *U* and *V*. Further, *W* should not be arbitrary but should be determinable from *U* and *V* in a principled manner.

Consider the proto-rule

$$\frac{x : T \qquad y : U \qquad z : V \qquad T <: \mathit{Int} \qquad \text{combinable?}(U, V)}{\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z\ \texttt{endif} : \text{combine}(U, V)}$$

"combinable?(U, V)" should be taken as an expression that is true if and only if types U and V can be sensibly combined. "combine(U, V)" should be taken as an expression standing for the type W that results from combining U and V.

The simplest possibility is to require the true- and false-clauses to have equal types and to let that type be the resultant type of the `if` expression; thus combinable?(U, V) would simplify to $U = V$, and combine(U, V) would become U (or V). While simple, this interpretation is quite rigid and can be extremely inconvenient for programmers (e.g., if U were `int`, and V were `immut int`, the `if` expression would be disallowed).

A better choice is to define combinable?(U, V) as compatibility (rather than equality) between types U and V (where, as above, the compatibility of U and V means that $U <: V$ or $V <: U$). While still simple, this interpretation allows programmers much more freedom. However, a suitable definition for combine(U, V) must be then determined.

One possibility is to define combine(U, V) as

combine(U, V) =
$\quad U$, if $V <: U$ and $\leftarrow (U <: V)$
$\quad V$, if $U <: V$ and $\leftarrow (V <: U)$
$\quad U$ (or, alternatively, V), if $U <: V$ and $V <: U$

While such a definition would be technically correct, the arbitrary choice of U (or V) in the last case has little justification. For example, if U is `immut con int` and V is `con immut int`, choosing either U or V is somewhat difficult to defend.

A more satisfactory answer is to construct type W from U and V so that W is, in some sense, as similar to both U and V as can be managed, while still ensuring that W is a supertype of both U and V. (Another possibility, not addressed here, is to combine the types in such a way that W is, in some sense, as simple as possible.) The dual concepts of *disjoined supertype* and *conjoined subtype* provide a reasonably simple definition of "as similar as can be managed."

For compatible types T and U, the disjoined supertype of T and U (written "dsup(T, U)") and the conjoined subtype of T and U (written "csub(T, U)") can be defined as

dsup(T, U) =
$\quad \mathit{Immut}$ dsup(V, W), if $T = \mathit{Immut}\ V$ and $U = \mathit{Immut}\ W$
$\quad \mathit{Immut}$ dsup(V, U), if $T = \mathit{Immut}\ V$ and mutable?(U)
$\quad \mathit{Immut}$ dsup(T, W), if mutable?(T) and $U = \mathit{Immut}\ W$
$\quad \mathit{Triv}$, if $T = \mathit{Triv}$ and $U = \mathit{Triv}$
$\quad \mathit{Int}$, if $T = \mathit{Int}$ and $U = \mathit{Int}$
$\quad \mathit{Ref}$ dsup(V, W), if $T = \mathit{Ref}\ V$ and $U = \mathit{Ref}\ W$
$\quad \mathit{Refvec}$ dsup(V, W), if $T = \mathit{Refvec}\ V$ and $U = \mathit{Refvec}\ W$
$\quad \mathit{Con}$ dsup(V, W), if $T = \mathit{Con}\ V$ and $U = \mathit{Con}\ W$
$\quad \mathit{Fun}$ (csub(V_1, X_1), … , csub(V_n, X_n)) dsup(W, Y),
$\qquad$ if $T = \mathit{Fun}\ (V_1, \ldots, V_n)\ W$ and $U = \mathit{Fun}\ (X_1, \ldots, X_n)\ Y$

csub(T, U) =
$\quad \mathit{Immut}$ csub(V, W), if $T = \mathit{Immut}\ V$ and $U = \mathit{Immut}\ W$
$\quad$ csub(V, U), if $T = \mathit{Immut}\ V$ and mutable?(U)
$\quad$ csub(T, W), if mutable?(T) and $U = \mathit{Immut}\ W$
$\quad \mathit{Triv}$, if $T = \mathit{Triv}$ and $U = \mathit{Triv}$

Int, if $T = \mathit{Int}$ and $U = \mathit{Int}$
$\mathit{Ref}\ \mathrm{csub}(V, W)$, if $T = \mathit{Ref}\ V$ and $U = \mathit{Ref}\ W$
$\mathit{Refvec}\ \mathrm{csub}(V, W)$, if $T = \mathit{Refvec}\ V$ and $U = \mathit{Refvec}\ W$
$\mathit{Con}\ \mathrm{csub}(V, W)$, if $T = \mathit{Con}\ V$ and $U = \mathit{Con}\ W$
$\mathit{Fun}\ (\mathrm{dsup}(V_1, X_1), \ldots, \mathrm{dsup}(V_n, X_n))\ \mathrm{csub}(W, Y)$,
if $T = \mathit{Fun}\ (V_1, \ldots, V_n)\ W$ and $U = \mathit{Fun}\ (X_1, \ldots, X_n)\ Y$

(Of note is the manner in which `fun` types are combined.)

Adopting these definitions, the rule corresponding to production (14e) becomes

$$\frac{x : T \qquad y : U \qquad z : V \qquad T <: \mathit{Int} \qquad \mathrm{compatible?}(U, V)}{\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z\ \texttt{endif} : \mathrm{dsup}(U, V)}$$

From production (14f):

$$\frac{x : \mathit{Con}\ T \qquad y : U \qquad U <: T}{\texttt{control}\ x\ \texttt{in}\ y\ \texttt{endcontrol} : T}$$

It might be argued that the type of a control block should be the type of its body; however, the value of a control block can be either the value of its body or the value coming from a `continues` expression. Letting the type of the control block be determined by the type of its continuation variable permits typing control blocks and `continues` expressions in a consistent manner.

From production (14h):

$$\frac{x : \mathit{Fun}\ (T_1, \ldots, T_n)\ U \qquad y_1 : V_1 \qquad \ldots \qquad y_n : V_n \qquad V_1 <: T_1 \qquad \ldots \qquad V_n <: T_n}{x\ (y_1,\ y_2,\ \ldots,\ y_n)\ : U}$$

$$\frac{x : \mathit{Immut}\ \mathit{Fun}\ (T_1, \ldots, T_n)\ U \qquad y_1 : V_1 \qquad \ldots \qquad y_n : V_n \qquad V_1 <: T_1 \qquad \ldots \qquad V_n <: T_n}{x\ (y_1,\ y_2,\ \ldots,\ y_n)\ : U}$$

A function application is meaningful only if the type of each actual parameter is a subtype of the type of the corresponding formal parameter. If the application is meaningful, its type is the type of the function's body.

From production (17b):

$$\frac{x : \mathit{Refvec}\ T \qquad y : U \qquad U <: \mathit{Int}}{x\,[y]\ : T}$$

$$\frac{x : \mathit{Immut}\ \mathit{Refvec}\ T \qquad y : U \qquad U <: \mathit{Int}}{x\,[y]\ : T}$$

The index operation can be used only with references to vectors, and the subscript must be an `int` or `immut int`.

From production (17c):

$$\frac{x : \mathit{Ref}\ T}{x\ \texttt{@} : T}$$

$$\frac{x : \mathit{Immut}\ \mathit{Ref}\ T}{x\ \texttt{@} : T}$$

From production (18a):

$$\frac{}{\mathit{integerConstant} : \mathit{Int}}$$

From production (18b):

$$\frac{}{\texttt{?} : \mathit{Triv}}$$

From production (19a):

$$\frac{x_1 : T_1 \quad \dots \quad x_n : T_n \quad y : U}{\texttt{vars}\ x_1\ id_1\texttt{,}\ \dots\ \texttt{,}\ x_n\ id_n\ \texttt{in}\ y\ \texttt{endvars} : U}$$

The type of a variable block is determined by the inferred type of its body. It is useful to compare the rule for variable blocks with the rule for function constants (production (2)).

From production (23a):

$$\frac{x : T \quad y : U \quad T <: \mathit{Int}}{\texttt{vec [}x\texttt{]}\ y : \mathit{Refvec}\ U}$$

Type Conversion

Type conversion is necessary in CCL whenever a value of type T is provided when a value of type U is expected; however, type conversion is trivial in CCL. If $T <: U$ and $T \neq U$, T and U can differ only with respect to their immutability and/or to the immutability of their components (e.g., `immut con int` vs. `con immut int`). This being the case, type conversion in CCL is simply a compile-time, type-checking point of view: no run-time work is ever required, since all immutable values are operationally identical to their mutable counterparts.

SAMPLE CCL PROGRAMS

The following programs (taken from a suite of test programs used to evaluate student CCL compilers) illustrate many of the basic and more advanced features of CCL.

A Factorial Program

The following CCL program determines n! for n = 0, ... , 10 using a typical recursive implementation of the factorial function:

```
fun ()

    vars

```

```
        fun (int) int fact,
        int j

    in

        !
        ! Let fact = the naive recursive factorial function.
        !

            fact :=

                fun (int n)
                    if n = 0 then
                        1
                    else
                        n * fact(n - 1)
                        endif
                    endfun;

        !
        ! For j = 0, ... , 10, calculate and display j!.
        !

            j := 0;
            while j <= 10 do
                output fact(j);
                j := j + 1
                endwhile

        endvars

    endfun
```

Running the program produces the obvious output:

```
   1
   1
   2
   6
   24
   120
   720
   5040
   40320
   362880
   3628800
```

A Continuation-Passing Factorial Program

The following CCL program determines $n!$ for $n = 0, \ldots , 10$ using a continuation-passing style (Friedman et al., 2008) implementation of the factorial function. (Note that the term "continuation" here does not refer to a CCL value of type `con` *T*, but to a "continuation" in sense of a function to be applied to a result.)

```
fun ()

    vars

        fun (int, fun (int) int) int fact,
        int j

    in

        !
        ! Let fact = the continuation-passing style factorial function.
```

```
        !

            fact :=

                fun (int n, fun (int) int k)

                    if n = 0 then

                        !
                        ! Apply the continuation to the result.
                        !

                            k(1)

                    else

                        !
                        ! Apply fact with a continuation that will (eventually)
                        ! multiply (n-1)! by n.
                        !

                            fact(n - 1, fun (int result) k(n * result) endfun)

                        endif

                    endfun;

        !
        ! For j = 0, ... , 10, calculate and display j!.
        !

            j := 0;
            while j <= 10 do

                !
                ! Calculate and display j! by applying fact to j and the
                ! identity function as actual parameters.
                !

                    output fact(j, fun (int n) n endfun);

                j := j + 1
                endwhile

        endvars

    endfun
```

This program produces the same output as the first version of the factorial program above.

A Factorial Program Using Continuations

The following CCL program demonstrates (in a contrived fashion) the use of CCL continuations (i.e., values of subtypes of type `con` *T*). This program recursively determines *n*! for *n* = 0, ... , 10 through the use of an "accumulator" parameter. However, while many function applications are performed, only the very outermost function ever performs a normal return.

```
fun ()

    vars

        fun (int, int, con int) int fact,
        int j,
        con int answerCon

    in
```

```

        !
        ! Let fact = the contrived factorial function, using an accumulator
        ! parameter and a continuation.
        !

            fact :=

                fun (int n, int answerValue, con int answerCon)
                    if n = 0 then

                        !
                        ! Produce the value of the accumulator parameter as the
                        ! value of the control block that created the continuation.
                        !

                            answerCon continues answerValue

                    else

                        !
                        ! Apply fact, accumulating the factor of n and passing along
                        ! the continuation through which to produce the final answer.
                        !

                            fact(n - 1, answerValue * n, answerCon)

                        endif

                    endfun;

        !
        ! For j = 0, ... , 10, calculate and display j!.
        !

            j := 0;
            while j <= 10 do

                !
                ! Calculate and display j! by applying fact to j, an initial
                ! accumulator value of 1, and a continuation through which to produce
                ! the final result.
                !

                    output
                        control
                            answerCon
                        in
                            fact(j, 1, answerCon)
                            endcontrol;

                j := j + 1
                endwhile

        endvars

    endfun
```

This program also produces the same output as the first version of the factorial program above.

A Program Illustrating Vectors

The following program illustrates a variety of language constructs via implementations of simple vector input, output, and right-associative reduction operations:

```
fun ()

```

```
vars

    int n,
    vec [ input n ] int v,
    int reduction,
    fun (ref vec int) triv readIntVector,
    fun (ref vec immut int) triv writeIntVector,
    fun (ref vec int, ref int, fun (int, int) int, int) triv reduceIntVector

in

    !
    ! Let readIntVector = a function that reads values into v[ 0 ], ... ,
    ! v[ # v - 1 ], successively.
    !

        readIntVector :=

            fun (ref vec int v)

                vars

                    int j

                in

                    j := 0;
                    while j < # v do
                        input v[ j ];
                        j := j + 1
                        endwhile

                    endvars;

                ?

                endfun;

    !
    ! Let writeIntVector = a function that displays v[ 0 ], ... , v[ # v - 1 ],
    ! successively.
    !

        writeIntVector :=

            fun (ref vec immut int v)

                vars

                    int j

                in

                    j := 0;
                    while j < # v do
                        output v[ j ];
                        j := j + 1
                        endwhile

                    endvars;

                ?

                endfun;

    !
    ! Let reduceIntVector = a function that assigns to the variable pointed at
    ! by result the value
    !
```

```
        !    f(v[ 0 ],
        !        f(v[ 1 ],
        !            ...
        !                f(v[ # v - 1 ],
        !                    vacuous) ... ))
        !

            reduceIntVector :=

                fun (
                        ref vec immut int v,
                        ref int result,
                        fun (int, int) int f,
                        int vacuous
                        )

                    vars

                        int j

                    in

                        j := # v - 1;
                        result @ := vacuous;
                        while j >= 0 do
                            result @ := f(v[ j ], result @);
                            j := j - 1
                            endwhile

                        endvars;

                    ?

                    endfun;

        readIntVector(v);
        writeIntVector(v);

        !
        ! Let reduction = the product of v[ 0 ] ... v[ # v - 1 ]
        !

            reduceIntVector(v, & reduction, fun (int j, int k) j * k endfun, 1);

        output reduction;

        !
        ! Let reduction = the sum of v[ 0 ] ... v[ # v - 1 ]
        !

            reduceIntVector(v, & reduction, fun (int j, int k) j + k endfun, 0);

        output reduction

        endvars

    endfun
```

A sample execution of this program produces the following input/output (with the user's input shown underlined and annotations shown italicized):

```
    5                              (the value to be read in as the size of the vector)
    -1  2  -3  4  -5               (the values to be read into the vector elements)
    -1                             (the values in the vector elements)
    2
    -3
    4
```

```
-5
-120                                    (the reduction of the vector through multiplication)
-3                                      (the reduction of the vector through addition)
```

COMPARISON WITH OTHER SUBJECT LANGUAGES

Many texts on compilers present subject languages for use in compilers courses, either in extended examples used throughout the text or in individual chapters or appendices describing specific projects. Most texts present subsets of existing programming languages. Choices include subsets of Pascal-related languages, subsets of C-related languages, subsets of Ada, and subsets of various object-oriented languages. Some texts present quite rich subject languages, with the issue of subsetting left to the instructor.

Choosing a subset of an existing language can be attractive, for a variety of reasons (e.g., students can rely on previous experience with the language, and existing compilers can be used for comparison and testing). However, it can be quite difficult to create a subset that is small enough to be manageable but contains only "useful" or "interesting" features. Further, there may be no existing language containing all the features one considers "interesting."

CCL is designed around a small imperative core having a few basic types, a small collection of expressions, the basic control structures (sequential execution, iterative execution, and conditional execution), and functions. Added on top of this core, in a reasonably orthogonal manner, are pointers, local variables, dynamically-sized vectors, and continuations. While specific inclusions and omissions can certainly be argued, this particular collection of features, it is felt, provides students with a tractable, yet satisfying, challenge, one that brings together concepts from a number of classic and contemporary programming languages.

USING CCL IN A COMPILERS COURSE

The Course and the Students

The author has used CCL as a basis for projects in a one-semester, elective compilers course for a number of years. The course has as prerequisites

- a first-year, one-semester course in data structures and design using the C++ programming language
- a second-year, one-semester course in computer organization (which includes assembly language programming)
- a second-year, one-semester course in various paradigms of computation (including functional programming and object-oriented programming)
- a third-/fourth-year, one-semester course in automata, formal languages, and computability theory

Class time is spent on lectures and discussion, and there is no formal laboratory component to the course. Various primary texts, including Aho et al., 2006, Appel, 1998, Cooper et al, 2011, and Fischer et al, 2009, have been used over the years. While the course is not designated as a "capstone" course, it is viewed that way by a number of faculty and students.

The course is taken almost exclusively by computer science majors in their third or fourth year of study. Roughly half of the students have taken a third-/fourth-year course in analysis of algorithms prior to taking the compilers course. Some of the students have previous familiarity writing definitional interpreters for programming languages (Friedman et al., 2008). The students generally do not have previous experience using the necessary specialized software tools (e.g., parser generators and scanner generators) or working on a program design and development project of comparable size.

The Compiler Project

A typical semester-long project involves the students, individually, using the C or C++ programming language, the Yacc (Johnson, 1975) or Bison (Donnelly & Stallman, 2002) parser generator, and the Lex (Lesk, 1975) or Flex (Paxson, 1995) scanner generator to implement a multiple-pass compiler for (nearly) the complete CCL language. Trees serve as the intermediate language of the compiler, and assembly language (typically Intel 80x86 assembly language) serves as the target language. Student compilers usually generate relatively naive assembly language that is linked with a C run-time library that provides support for input/output.

A straightforward, stack-based run-time model is used. An application of a function value or an execution of a variable block creates a new activation record (or "stack frame") on the top of the run-time stack. The block-structured data environment is maintained through the use of a display stored within the activation record. A function-constant value can then be represented as a pair: the address of the first instruction of the body of the function and the address of the activation record that was on top of the stack when the function value was created. A continuation can be represented as a triple: the address of the instruction following the control block for which the continuation was created, the address of the activation record that was on top of the stack when the continuation was created, and the value of the top-of-stack pointer at the time the continuation was created. Since vectors are represented indirectly in CCL, the exact offsets of formal parameters and local variables within activation records are easily calculated at compile time.

The project proceeds in phases:

- implementation of an unadorned parser and scanner (based on the grammar presented in Appendix A)
- implementation of support code for representing and manipulating abstract-syntax trees, symbol tables, and so on (based on a rudimentary C++ implementation of *n*-ary trees supplied by the instructor)
- implementation of tree generation
- implementation of type-checking
- development of a run-time stack model
- implementation of code generation for simple expressions and control structures
- implementation of code generation for function constants and function application
- implementation of code generation for variable blocks
- implementation of code generation for one-dimensional vectors
- implementation of code generation for control blocks and continuations

At each step, the students' results are evaluated against suites of test programs (supplied by the instructor) which are subsequently made available to the students. At all times, students have access to executable versions of a sample compiler and, more importantly, a definitional interpreter (implemented in R6RS-compliant Scheme (Sperber at al., 2007)) that gives a precise semantics for CCL and provides answers to specific questions students may have concerning the meaning of CCL constructs.

The project can be simplified by omitting various language features: immutable types; vectors; control blocks and continuations; and/or variable blocks. Students are sometimes asked to extend the language in simple ways. Student compilers usually range in size from approximately 4000 to 7000 lines (excluding documentation) of C/C++ code, Yacc/Bison code, and Lex/Flex code.

Experiences

The author's experiences using CCL in compilers courses have generally been quite positive. Considering all the students who have taken the compilers course, approximately one-third have completed (or very nearly completed) the project, one-third have produced compilers capable of handling a reduced subset of CCL, and one-third have produced compilers that functioned to a lesser degree. The most common difficulties students have encountered can be grouped into three categories: difficulties with project management, difficulties mastering the semantics of CCL itself, and difficulties mapping the semantics of CCL onto the compile-time and run-time environments.

Difficulties managing the project have generally stemmed from students' lack of experience managing such a large project. Students discover, relatively early, that the time-management, program-organization, and program-testing techniques that served them adequately on small projects often do not scale up well to a large project. During the second phase of the project (implementation of support code for representing and manipulating abstract-syntax trees, symbol tables, etc.), most students develop adequate management regimens; those students who do not develop such regimens are at risk for not completing the project. Project management tools and techniques are discussed in lecture as needed.

Difficulties mastering CCL itself have generally involved aspects of CCL functions (e.g., that CCL functions are anonymous values and that CCL permits higher-order functions), CCL continuations, and the relatively general manner in which CCL constructs can be nested. Through a series of carefully chosen programming projects using CCL given at strategic points in the semester, students generally master this material with relative ease. While students having familiarity with programming languages having first-class functions (e.g., Scheme and ML (Milner et al., 1997)) and languages having relatively regular nesting properties (e.g., Pascal and ML) are at something of an advantage initially, most students having adequate experience learning new programming languages master the concepts in time.

Most of the difficulties students have encountered in mapping the semantics of CCL onto the compile-time and run-time environments have been classic issues related to lexical scope rules, strict typing, block structure, pointer types, and so forth. For CCL, students also face the difficulties of dealing with function constants, continuations, and dynamically-sized vectors.

EXTENDING CCL

More sophisticated projects might involve generating more optimal code, adding features to the language, or modifying the semantics of the language. These extensions range from quite simple to quite complex. Excellent projects concerning code optimization are available from many texts, including Aho et al., 2006, and Fischer et al, 2009. Projects concerning modifications or extensions to CCL include

- generating code to perform various run-time checks (checks for division by zero, bounds checking for vector subscripts, etc.)
- implementing vectors of vectors (typically based on classic techniques like those found in Randell & Russell, 1964, and Sattley, 1961)

- adding a Boolean type or a character type, including appropriate constants and operators. A key decision is whether the new type should be related to type `int` (as in C and C++) or completely distinct from type `int` (as in Pascal or Ada).
- adding a mechanism for initializing immutable local variables
- adding a heterogeneous ordered-pair type
- permitting certain kinds of expressions (e.g., conditional expressions and expression sequences) to be used used in contexts where l-values are required
- adding a mechanism for dynamically-created, heap-allocated variables, either explicitly destroyed or garbage-collected (with run-time support from appropriate libraries)
- adding a heterogeneous *n*-tuple type having named components (similar to Pascal records or C structures)
- adding a mechanism for declaring and using symbolic constants. An approach consistent with basic CCL is to introduce *constant block* expressions of the form

  `consts` *constantDeclarationList* `in` *expressionSequence* `endconsts`

  Symbolic constants cannot be assigned to via `:=` nor be referenced via `&`. The introduction of symbolic constants raises a number of issues related to type-checking and code generation, as well as requiring the introduction of a mechanism for initializing constants.
- adding a mechanism for declaring and using named types. An approach consistent with basic CCL is to introduce *type block* expressions of the form

  `types` *variableDeclarationList* `in` *expressionSequence* `endtypes`

  (Various issues exist regarding identifying and resolving relevant differences between the `vec` and `ref vec` types in the presence of named types, comparing named types, etc.)
- generating executable code that is properly tail-recursive (Friedman et al., 2008)

The implementation of run-time checks and simple code optimizations (e.g., constant folding) are quite easy extensions to the project, which a number of students are able to complete in a one-semester course. The more difficult optimizations or extensions have sometimes been undertaken by students as independent-study projects.

Even more advanced, more extensive projects might include adding object-oriented features, a module mechanism, first-class functions, and other features to the language.

ACKNOWLEDGEMENTS

The author wishes to thank the referees of Zaring, 2001, for their many thoughtful comments and suggestions.

REFERENCES

Aho, A., Lam, M., Sethi, R., & Ullman, J. (2006). *Compilers: principles, techniques, and tools* (2nd ed.). Reading, MA: Addison Wesley.

Appel, A. (1998). *Modern compiler implementation in C*. New York, NY: Cambridge University Press.

Appel, A. (2002). *Modern compiler implementation in Java* (2nd ed.). New York, NY: Cambridge University Press.

Cooper, K., & Torczon, L. (2011). *Engineering a compiler* (2nd ed.). Burlington, MA. Morgan Kaufmann.

Demers, A., & Teitelbaum, R. (1980). *The Toy programming language*. Unpublished manuscript, Cornell University.

Donnelly, C., & Stallman, R. (2002). *Bison: the YACC-compatible parser generator*. Boston, MA: Free Software Foundation.

Fischer, C., Cytron, R., & Leblanc, R. (2009) *Crafting a compiler*. New York, NY: Pearson.

Flatt, M., and PLT (2016). *The Racket Reference, Version 6.7*. `docs.racket-lang.org/reference/index.html`.

Friedman, D., & Wand, M. (2008). *Essentials of programming languages* (3rd ed.). Cambridge, MA: MIT Press.

Grune, D., van Reeuiwjk, K., Bal, H., Jacobs, C., & Langendoen, K. (2012). *Modern Compiler* Design (2nd ed.). New York, NY: John Wiley & Sons.

Holub, A. (1990). *Compiler design in C*. Englewood Cliffs, NJ: Prentice-Hall.

Jensen, K., Wirth, N., & Mickel, A. (1992). *Pascal user manual and report* (4th ed.). New York, NY: Springer Verlag.

Johnson, S. (1975). Yacc: yet another compiler compiler. *Computing Science Technical Report 32*. Murray Hill, NJ: Bell Laboratories.

Kernighan, B., & Ritchie, D. (1988). *The C programming language* (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall.

Lesk, M. (1975). Lex: a lexical analyzer generator. *Computing Science Technical Report 39*. Murray Hill, NJ: Bell Laboratories.

Mak, R. (2009). *Writing compilers and interpreters* (3rd ed.). New York, NY: John Wiley & Sons.

Milner, R., Tofte, M., Harper, R., & MacQueen, D. (1997). *The definition of Standard ML (revised)*. Cambridge, MA: MIT Press.

Moses, J. (1970). The function of FUNCTION in LISP, or why the FUNARG problem should be called the environment problem. *AI Memo 199*. Cambridge, MA: MIT Artificial Intelligence Laboratory.

Paxson, V. (1995). *Flex: a fast scanner generator*. Berkeley, CA: Lawrence Berkeley Laboratory.

Pierce, B. (2002). *Types and programming languages*. Cambridge, MA: MIT Press.

Plauger, P. (1992). *The standard C library*. Englewood Cliffs, NJ: Prentice-Hall.

Randell, B., & Russell, L. (1964). *ALGOL 60 implementation: the translation and use of ALGOL 60 programs on a computer*. New York, NY: Academic Press.

Sattley, K. (1961). Allocation of storage for arrays in ALGOL 60. *Communications of the ACM, 4*(1).

Scott, M. (2015). *Programming language* pragmatics (4th ed.). Burlington, MA: Morgan Kaufmann.

Sperber, M., Dybvig, R., Flatt, M., & van Straaten, A. (Eds.) (1998). Revised^6 report on the algorithmic language Scheme. Scheme Steering Committee.

Steele, G. (1990). *Common LISP: the language* (2nd ed.). Newton, MA: Digital Press.

Stroustrup, B. (2013). *The C++ programming language* (4th ed.). Reading, MA: Addison Wesley.

van Wijngaarden, A., Mailloux, B., Peck, J., Koster, C., Sintzoff, M., Lindsey, C., Meertens, L., & Fisker, R. (1975). Revised report on the algorithmic language Algol 68. *Acta Informatica 5*, 1-236.

Watt, D., & Brown, D. (2000). *Programming language processors in Java*. New York, NY: Prentice-Hall.

Wilhelm, R., Seidl, H., & Hack, S. (2013). *Compiler design: syntactic and semantic analysis*. New York, NY: Springer.

Wirth, N. (1996). *Compiler construction*. Reading, MA: Addison Wesley.

Zaring, A. (2001). CCL: a subject language for compiler projects. *Computer Science Education, 11*(2).

APPENDIX A: THE CCL GRAMMAR

(1) *program*
(1a) ::= *functionConstant*

(2) *functionConstant*
(2a) ::= `fun` `(` *formalParameterDeclarationPart* `)` *expressionSequence*
`endfun`

(3) *formalParameterDeclarationPart*
(3a) ::= ε
(3b) | *formalParameterDeclarationList*

(4) *formalParameterDeclarationList*
(4a) ::= *formalParameterDeclaration*
(4b) | *formalParameterDeclarationList* `,` *formalParameterDeclaration*

(5) *formalParameterDeclaration*
(5a) ::= *formalParameterType* *identifier*

(6) *formalParameterType*
(6a) ::= *unqualifiedFormalParameterType*
(6b) | `immut` *unqualifiedFormalParameterType*

(7) *unqualifiedFormalParameterType*
(7a) ::= `triv`
(7b) | `int`
(7c) | `ref` *formalParameterType*
(7d) | `ref vec` *formalParameterType*
(7e) | `con` *formalParameterType*
(7f) | `fun` `(` *formalParameterTypePart* `)` *formalParameterType*

(8) *formalParameterTypePart*
(8a) ::= ε
(8b) | *formalParameterTypeList*

(9) *formalParameterTypeList*
(9a) ::= *formalParameterType*
(9b) | *formalParameterTypeList* `,` *formalParameterType*

(10) *expressionSequence*
(10a) ::= *expression*
(10b) | *expressionSequence* `;` *expression*

(11) *expression*
(11a) ::= *assignmentExpression*
(11b) | *assignmentExpression* `continues` *expression*

(12) *assignmentExpression*
(12a) ::= *simpleExpression*
(12b) | *variableExpression* `:=` *assignmentExpression*

(Note: For the sake of brevity, productions (13a)-(13q) are ambiguous and do not capture the precedences and associativities of the operators as described earlier.)

(13) *simpleExpression*
(13a) ::= *primaryExpression*
(13b) | *simpleExpression* `=` *simpleExpression*
(13c) | *simpleExpression* `<>` *simpleExpression*
(13d) | *simpleExpression* `<=` *simpleExpression*
(13e) | *simpleExpression* `<` *simpleExpression*
(13f) | *simpleExpression* `>=` *simpleExpression*
(13g) | *simpleExpression* `>` *simpleExpression*
(13h) | *simpleExpression* `+` *simpleExpression*
(13i) | *simpleExpression* `-` *simpleExpression*
(13j) | *simpleExpression* `*` *simpleExpression*
(13k) | *simpleExpression* `/` *simpleExpression*
(13l) | `input` *variableExpression*
(13m) | `output` *simpleExpression*
(13n) | `+` *simpleExpression*
(13o) | `-` *simpleExpression*
(13p) | `#` *simpleExpression*
(13q) | `&` *variableExpression*

(14) *primaryExpression*
(14a) ::= *constant*
(14b) | *variableExpression*
(14c) | `(` *expressionSequence* `)`
(14d) | `while` *expressionSequence* `do` *expressionSequence* `endwhile`
(14e) | `if` *expressionSequence* `then` *expressionSequence* `else` *expressionSequence* `endif`
(14f) | `control` *variableExpression* `in` *expressionSequence* `endcontrol`
(14g) | *variableBlock*
(14h) | *primaryExpression* `(` *actualParameterPart* `)`

(15) *actualParameterPart*
(15a) ::= ε
(15b) | *actualParameterList*

(16) *actualParameterList*
(16a) ::= *expressionSequence*
(16b) | *actualParameterList* `,` *expressionSequence*

(17) *variableExpression*
(17a) ::= *identifier*
(17b) | *primaryExpression* `[` *expressionSequence* `]`
(17c) | *primaryExpression* `@`

(18) *constant*
(18a) ::= *integerConstant*
(18b) | `?`
(18c) | *functionConstant*

(19) *variableBlock*
(19a) ::= `vars` *variableDeclarationList* `in` *expressionSequence* `endvars`

(20) *variableDeclarationList*
(20a) ::= *variableDeclaration*
(20b) | *variableDeclarationList* `,` *variableDeclaration*

(21) *variableDeclaration*
(21a) ::= *variableType identifier*

(22) *variableType*
(22a) ::= *formalParameterType*
(22b) | *unqualifiedVariableType*
(22c) | `immut` *unqualifiedVariableType*

(23) *unqualifiedVariableType*
(23a) ::= `vec` `[` *expressionSequence* `]` *variableType*

(24) *identifier*
(24a) ::= ...

(25) *integerConstant*
(25a) ::= ...

APPENDIX B: THE CCL SUBTYPE AND TYPE RELATIONS

The <: Relation

$$\frac{}{\mathit{Triv} <: \mathit{Triv}}$$

$$\frac{}{\mathit{Int} <: \mathit{Int}}$$

$$\frac{T <: U}{T <: \mathit{Immut}\ U}$$

$$\frac{T <: U}{\mathit{Immut}\ T <: U}$$

$$\frac{T <: U \qquad \text{mutable?}(T) \qquad \text{mutable?}(U)}{\mathit{Ref}\ T <: \mathit{Ref}\ U}$$

$$\frac{T <: U \qquad \text{immutable?}(U)}{\mathit{Ref}\ T <: \mathit{Ref}\ U}$$

$$\frac{T <: U \qquad \text{mutable?}(T) \qquad \text{mutable?}(U)}{\mathit{Refvec}\ T <: \mathit{Refvec}\ U}$$

$$\frac{T <: U \qquad \text{immutable?}(U)}{\mathit{Refvec}\ T <: \mathit{Refvec}\ U}$$

$$\frac{T <: U}{\mathit{Con}\ T <: \mathit{Con}\ U}$$

$$\frac{V_1 <: T_1 \qquad V_2 <: T_2 \qquad \ldots \qquad V_j <: T_j \qquad U <: W}{\mathit{Fun}\ (T_1,\ \ldots\ ,\ T_j)\ U <: \mathit{Fun}\ (V_1,\ \ldots\ ,\ V_j)\ W}$$

For convenience only, rules expressing the reflexivity and transitivity of the <: relation are also included:

$$\frac{}{T <: T}$$

$$\frac{T <: U \qquad U <: V}{T <: V}$$

The : Relation

From production (2):

$$\frac{x_1 : T_1 \quad \ldots \quad x_n : T_n \quad y : U}{\texttt{fun}\ (x_1\ id_1\texttt{,}\ \ldots\ x_n\ id_n)\ y\ \texttt{endfun} : \mathit{Fun}\ (T_1, \ldots, T_n)\ U}$$

From production (6):

$$\frac{x : T}{\texttt{immut}\ x : \mathit{Immut}\ T}$$

From production (7a):

$$\frac{}{\texttt{triv} : \mathit{Triv}}$$

From production (7b):

$$\frac{}{\texttt{int} : \mathit{Int}}$$

From production (7c):

$$\frac{x : T}{\texttt{ref}\ x : \mathit{Ref}\ T}$$

From production (7d):

$$\frac{x : T}{\texttt{ref vec}\ x : \mathit{Refvec}\ T}$$

From production (7e):

$$\frac{x : T}{\texttt{con}\ x : \mathit{Con}\ T}$$

From production (7f):

$$\frac{x_1 : T_1 \quad \ldots \quad x_n : T_n \quad y : U}{\texttt{fun}\ (x_1\texttt{,}\ \ldots\ x_n)\ y : \mathit{Fun}\ (T_1, \ldots, T_n)\ U}$$

From production (10):

$$\frac{x_1 : T_1 \quad x_2 : T_2 \quad \ldots \quad x_n : T_n}{x_1\ \texttt{;}\ x_2\ \texttt{;}\ \ldots\ \texttt{;}\ x_n : T_n}$$

From production (11b):

$$\frac{x : \mathit{Con}\ T \quad y : U \quad U <: T}{x \texttt{ continues } y : T}$$

$$\frac{x : \mathit{Immut}\ \mathit{Con}\ T \quad y : U \quad U <: T}{x \texttt{ continues } y : T}$$

From production (12b):

$$\frac{x : T \quad y : U \quad \text{mutable?}(T) \quad U <: T}{x := y : T}$$

From productions (13b)-(13g):

$$\frac{x : T \quad y : U \quad T <: \mathit{Triv} \quad U <: \mathit{Triv}}{x\ op\ y : \mathit{Int}}$$

$$\frac{x : T \quad y : U \quad T <: \mathit{Int} \quad U <: \mathit{Int}}{x\ op\ y : \mathit{Int}}$$

where *op* is any of <, <=, >, >=, =, or <>.

$$\frac{x : T \quad y : U \quad T <: \mathit{Ref}\ V \quad \text{compatible?}(T, U)}{x\ op\ y : \mathit{Int}}$$

$$\frac{x : T \quad y : U \quad T <: \mathit{Refvec}\ V \quad \text{compatible?}(T, U)}{x\ op\ y : \mathit{Int}}$$

$$\frac{x : T \quad y : U \quad T <: \mathit{Con}\ V \quad \text{compatible?}(T, U)}{x\ op\ y : \mathit{Int}}$$

where *op* is = or <>.

From productions (13h)-(13k):

$$\frac{x : T \quad y : U \quad T <: \mathit{Int} \quad U <: \mathit{Int}}{x\ op\ y : \mathit{Int}}$$

where *op* is any of `+`, `-`, `*`, or `/`.

From production (13l):

$$\frac{x : \mathit{Int}}{\texttt{input } x : \mathit{Int}}$$

From production (13m):

$$\frac{x : T \quad T <: \mathit{Int}}{\texttt{output}\ x : \mathit{Int}}$$

From productions (13n)-(13o):

$$\frac{x : T \quad T <: \mathit{Int}}{op\ x : \mathit{Int}}$$

where *op* is unary + or unary −.

From production (13p):

$$\frac{x : T \quad T <: \mathit{Refvec}\ U}{\#\ x : \mathit{Int}}$$

From production (13q):

$$\frac{x : T}{\texttt{\&}\ x : \mathit{Ref}\ T}$$

From production (14c):

$$\frac{x : T}{\texttt{(}\ x\ \texttt{)}\ : T}$$

From production (14d):

$$\frac{x : T \quad y : U \quad T <: \mathit{Int}}{\texttt{while}\ x\ \texttt{do}\ y\ \texttt{endwhile} : \mathit{Int}}$$

From production (14e):

$$\frac{x : T \quad y : U \quad z : V \quad T <: \mathit{Int} \quad \text{compatible?}(U, V)}{\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z\ \texttt{endif} : \text{dsup}(U, V)}$$

From production (14f):

$$\frac{x : \mathit{Con}\ T \quad y : U \quad U <: T}{\texttt{control}\ x\ \texttt{in}\ y\ \texttt{endcontrol} : T}$$

From production (14h):

$$\frac{x : \mathit{Fun}\ (T_1, \dots, T_n)\ U \quad y_1 : V_1 \quad \dots \quad y_n : V_n \quad V_1 <: T_1 \quad \dots \quad V_n <: T_n}{x\ \texttt{(}y_1\texttt{,}\ y_2\texttt{,}\ \dots\ \texttt{,}\ y_n\texttt{)}\ : U}$$

$$\frac{x : \mathit{Immut}\ \mathit{Fun}\ (T_1, \dots, T_n)\ U \quad y_1 : V_1 \quad \dots \quad y_n : V_n \quad V_1 <: T_1 \quad \dots \quad V_n <: T_n}{x\ \texttt{(}y_1\texttt{,}\ y_2\texttt{,}\ \dots\ \texttt{,}\ y_n\texttt{)}\ : U}$$

From production (17b):

$$\frac{x : \mathit{Refvec}\ T \qquad y : U \qquad U <: \mathit{Int}}{x\,\texttt{[}y\texttt{]} : T}$$

$$\frac{x : \mathit{Immut}\ \mathit{Refvec}\ T \qquad y : U \qquad U <: \mathit{Int}}{x\,\texttt{[}y\texttt{]} : T}$$

From production (17c):

$$\frac{x : \mathit{Ref}\ T}{x\,\texttt{@} : T}$$

$$\frac{x : \mathit{Immut}\ \mathit{Ref}\ T}{x\,\texttt{@} : T}$$

From production (18a):

$$\frac{}{\mathit{integerConstant} : \mathit{Int}}$$

From production (18b):

$$\frac{}{\texttt{?} : \mathit{Triv}}$$

From production (19a):

$$\frac{x_1 : T_1 \quad \ldots \quad x_n : T_n \qquad y : U}{\texttt{vars}\ x_1\ id_1\texttt{,}\ \ldots\ \texttt{,}\ x_n\ id_n\ \texttt{in}\ y\ \texttt{endvars} : U}$$

From production (23a):

$$\frac{x : T \qquad y : U \qquad T <: \mathit{Int}}{\texttt{vec [}x\texttt{]}\ y : \mathit{Refvec}\ U}$$

The Disjoined Supertype and Conjoined Subtype

For compatible types T and U,

dsup(T, U) =

- Immut dsup(V, W), if $T = \mathit{Immut}\ V$ and $U = \mathit{Immut}\ W$
- Immut dsup(V, U), if $T = \mathit{Immut}\ V$ and mutable?(U)
- Immut dsup(T, W), if mutable?(T) and $U = \mathit{Immut}\ W$
- Triv, if $T = \mathit{Triv}$ and $U = \mathit{Triv}$
- Int, if $T = \mathit{Int}$ and $U = \mathit{Int}$
- Ref dsup(V, W), if $T = \mathit{Ref}\ V$ and $U = \mathit{Ref}\ W$
- Refvec dsup(V, W), if $T = \mathit{Refvec}\ V$ and $U = \mathit{Refvec}\ W$
- Con dsup(V, W), if $T = \mathit{Con}\ V$ and $U = \mathit{Con}\ W$

$\mathit{Fun}\ (\mathrm{csub}(V_1, X_1), \ldots, \mathrm{csub}(V_n, X_n))\ \mathrm{dsup}(W, Y)$,
  if $T = \mathit{Fun}\ (V_1, \ldots, V_n)\ W$ and $U = \mathit{Fun}\ (X_1, \ldots, X_n)\ Y$

csub(T, U) =
  $\mathit{Immut}\ \mathrm{csub}(V, W)$, if $T = \mathit{Immut}\ V$ and $U = \mathit{Immut}\ W$
  $\mathrm{csub}(V, U)$, if $T = \mathit{Immut}\ V$ and mutable?(U)
  $\mathrm{csub}(T, W)$, if mutable?(T) and $U = \mathit{Immut}\ W$
  Triv, if $T = \mathit{Triv}$ and $U = \mathit{Triv}$
  Int, if $T = \mathit{Int}$ and $U = \mathit{Int}$
  $\mathit{Ref}\ \mathrm{csub}(V, W)$, if $T = \mathit{Ref}\ V$ and $U = \mathit{Ref}\ W$
  $\mathit{Refvec}\ \mathrm{csub}(V, W)$, if $T = \mathit{Refvec}\ V$ and $U = \mathit{Refvec}\ W$
  $\mathit{Con}\ \mathrm{csub}(V, W)$, if $T = \mathit{Con}\ V$ and $U = \mathit{Con}\ W$
  $\mathit{Fun}\ (\mathrm{dsup}(V_1, X_1), \ldots, \mathrm{dsup}(V_n, X_n))\ \mathrm{csub}(W, Y)$,
  if $T = \mathit{Fun}\ (V_1, \ldots, V_n)\ W$ and $U = \mathit{Fun}\ (X_1, \ldots, X_n)\ Y$