Multi-language, Multi-target Compiler Development: Evolution of the Gardens Point Compiler Project

K.John Gough
Queensland University of Technology, Brisbane, Australia

Abstract. The Gardens Point project started in 1987 as an attempt to make Modula-2 available on a variety of 32-bit Unix platforms. Since that time, it has evolved into a flexible platform for research into compiler construction and language implementation. Gardens point compilers are available on about a dozen platforms, and are mostly freeware. Currently, the gardens point infrastructure is used for a variety of research projects in areas such as code selection, global optimization, register allocation, and exception handling. There are also a variety of language research issues which are being pursued by the group. This paper sets out a brief history of the project, and describes the insights which have come from the implementation of a variety of languages on most of the major current computer architectures.

1 Introduction: Original Goals

The history of the gardens point project began in the mid 1980s, when QUT’s School of Computing Studies decided to move a large part of their teaching onto UNIX servers. Until that point, teaching relied on PCs in the first year, and VAX and PDP-11 machines thereafter. We were already using Modula-2 for introductory programming subjects, using Dave Moore’s fast single-pass FTL compiler on the PCs, and the ETH five-pass M22 compiler for the PDP-11. By this time the author was maintaining the M22 compiler, ETH having moved their own compiler development onto Lilith machines.

As it turned out, the School decided to purchase the then very new HP Precision Architecture “RISC” machines. It soon became clear that the only way to get a Modula-2 compiler for such a machine was to create one for oneself. The opportunity to create an entirely new compiler family targeted on the emerging 32-bit UNIX machines proved irresistible to the author, and to another old compiler hand, John Hynd. We decided to call the compiler gardens point modula-2, since QUT is sited at beautiful “Gardens Point” on the Brisbane river, in the heart of the City of Brisbane.

The main attraction, at first, was to create a complete compiler which could be entirely new. Thus the compiler would use no code or even architectural concepts from the various compilers on which we had previously worked. Instead, we would do a design which, from the start, would take advantage of the 32-bit flat virtual address spaces of the new generation of machines. Thus we planned
to be free from what we saw as the tyranny of narrow address spaces. In retrospect it is clear that taking this approach allows for the design of a very clean software architecture, but that there are dangers in going too willingly down the inflationary path of memory gluttony. In our current generation of compilers, register allocation for certain pathological functions in the SPEC-92 benchmark suite causes the dynamic allocation of a single data structure which demands 3-Mb. Anyone who remembers writing a complete compiler which fitted in 56Kb is bound to slight nervousness when faced with such a statistic.

Another prime goal was to create a compiler which would set new standards for user-friendliness in the area of compile-time and runtime diagnostics. We were simply dismayed at the prevailing standard of error messages which compilers produced. Most Modula-2 compilers were better than most C compilers, for sound technical reasons, but were still rather primitive. Consider the following Modula-2-specific example

When a programmer imports two separate enumeration types, (say Languages and Islands) into a module, it is an unfriendly act for a compiler to respond

\begin{verbatim}
FROM Atlas IMPORT Islands;
**** -- doubly declared identifier
\end{verbatim}

What the user needs to know is that the identifier “java” appears as a value in both enumerations. The compiler knows the identity of the offending identifier, why not tell the user?

The dedication to better error reporting is not without cost, of course. In the case of static semantic errors, additional information needs to be propagated along the call chain. Thus the failure to insert a new identifier into a name scope cannot precipitate an immediate, generic diagnostic, but must be propagated back to a point where the declarative context is known.

In the case of runtime errors, \texttt{gpm} also makes valiant efforts. In the case of range errors, the \texttt{gpm} runtime produces a diagnostic which reports the out-of-range value, the limits of the permitted range, and the module and source line number at which the error occurred. With care, all of this can be done with no time penalty. All error jumps are vectored through intermediate locations where they pick up their individual information. This behaviour has a moderate space cost, and in very large programs may lead to somewhat worse paging behaviour.

1.1 The “via-C” compilers

The original compiler was designed to construct an abstract syntax tree (\textit{AST}), for the whole of a compilation unit, in memory, at once. This choice was made to allow arbitrary intra-module attributes to be evaluated. Code was to be emitted by a recursive tree-walker, with the original target language being Hewlett-Packard’s proprietary version of “\textit{Ucode}” [22]. For about one year it seemed that HP would make the interface available for third party developers, subject
to the usual non-disclosure conditions. Eventually, negotiations broke down, due to what was said to be the rather unfamiliar legal territory into which such an agreement would have taken the company. Fortunately, we had been working on a contingency plan which involved traversal of the same AST to emit ANSI-C.

As shown in Figure 1, by substituting a different tree-walker a different output language could be produced. A small number of attributes were required for one tree-walker but not the other, but the majority of the effort was preserved.

The use of ANSI-C as an intermediate language is by now well established, but involves some rather tricky design considerations. One design issue which has seen a number of different attacks, is access to variables in non-local scopes (the so-called uplevel addressing problem). Some early designs, such as p2c allocate explicit activation records, so that explicit static links can be maintained. Currently, the most popular method to attack the problem seems to be to declare additional arguments in the C procedures, which explicitly pass references to each uplevel-accessed datum along the call chain. The “via-C” versions of gpm adopted an entirely different approach involving the maintenance of a version of Dijkstra’s display vector[8]. In the usual organisation, the elements of the display vector point to the base of the stack-frames of the accessible scopes. Since C has no explicit concept of activation record, we make the display vector point to an arbitrary uplevel-addressed datum in the frame. In the unusual case where there are several such data, we dynamically compute address differences, and hold these in statically allocated locations. This rather baroque method is surprisingly efficient. In the case of the common idiom in Figure 2 the resulting object code is as efficient as any native code implementation. Note that with our method, no extra cycles are consumed in the calls of Look, and the single call of Find attracts the overhead of a save and restore of a single display element. In this example, the display vector element would point directly to the variable

Fig. 1. Compiler frontend with multiple tree-walkers
PROCEDURE Find(key: INTEGER; Var ok: BOOLEAN);
PROCEDURE Look(k: INTEGER; t: Tree);
BEGIN
  IF t = NIL THEN ok := FALSE; RETURN END;
  IF k < t^.key THEN Look(k, t^.lOp)
  ELSIF k > t^.key THEN Look(k, t^.rOp)
  ELSE ok := TRUE; RETURN
END
END Look;
BEGIN (* Find *)
  Look(key, root)
END Find;

Fig. 2. Binary tree lookup, with non-local variable access

parameter ok.

Another difficulty is the mapping of *Modula-2* types into *ANSI-C* types. More recent compilers which translate languages with polymorphic assignment into *ANSI-C* have invariably abandoned all use of *C*’s type system, and use casts everywhere. We would have been well advised to do the same. Different *C* compilers ascribe quite different semantics to recursive type declarations in complex cases involving function prototypes. Furthermore, in *ANSI-C* there is no way to declare a recursive type which does not involve a *struct* or a *union*, *ISO Modula-2* [18], on the other hand, allows recursion through procedure types, and also through arrays. None of these constructs has an analog in *C*.

At one time, while we were still developing our native code compilers, we offered the “via-*C*” compiler on no less than eleven platforms. Some of these did not have *ANSI-C* conformant compilers, and almost all had bugs or idiosyncrasies which turned support into a major challenge. We survived by using a preprocessor which selectively included code to step around the known bugs of each *C* compiler.

1.2 The Dcode compilers

Apart from a few remaining “via-*C*” compilers, all versions of *gpm* now generate native code based on the use of an intermediate program representation, which is the instruction set of an abstract stack machine. We call the intermediate form *Dcode*, as it was designed to be used to express Directed acyclic graphs. In principle, the use of such a form enables a clean separation between the language dependent (and target independent) compiler front end, and the target dependent (and language independent) code-generating backend, as shown in Figure 3. Such an intermediate form is necessarily both language- and target-independent.

The use of abstract machine intermediate forms is the subject of much debate. One school of thought suggests that the separation of language and target
concerns should be realised by the generation of orthogonal specifications, and the composition of such specifications within a tool framework, such as Eli [17]. It is true that for any given, single language-and-target combination, a monolithic compiler can be made smaller and faster than a two-part compiler based on a language- and target-independent intermediate form, but the abstract machine based design still has much to recommend it in practice. Some of the advantages and disadvantages are commented on here.

The first of the $Dcode$ compilers was built for the $mips$ $R-3000$ architecture, with both big- and little-endian versions being created for $DecStation$ and $Silicon$ $Graphics$ machines. Later versions were created for $Intel$ $iapx86$, $SUN$ $SPARC$, $Digital$ $Alpha$ and $IBM$ $Power$ architectures.

1.3 Mixed compilation and interpretation

One of the benefits of adopting an abstract stack machine as an intermediate representation is that it opens up the possibility of interpretation as an implementation strategy. Indeed, the exploitation of just this possibility was the key idea behind the $UCSD$ $Pascal$ compilers, which first brought high level language programming to the world of microcomputers in the 1970s.

By 1990, at QUT our first-year students were still predominantly using PC machines for their programming, and many at that time were still using 8088 or 80286 processors. The possibility of creating a native code version of $gpm$ for these machines seemed remote. The design of $gpm$ specified the whole-number types as 32-bits, and the profligate use of memory in the frontend made the goal of bootstrapping a PC version unthinkable.

Instead, we translated the $Dcode$ version of the frontend into $C$ using the “via-C” compiler, and compiled it for the PC using $Turbo-C$. A vestigial backend performed some peephole compression of the $Dcode$, and cleverly combined this
with some compiled code to achieve mixed compilation and interpretation. This version, which we dubbed **gpm**-pc, used an interpreter to emulate a 32-bit machine, and included a software floating point implementation. Students using **gpm**-pc could develop their programs on a PC, and then run them on the UNIX machines with absolutely identical results, other than execution speed. On computationally intensive code, **gpm**-pc runs at 10–20% of the speed of the same machine running compiled code (but also using the long datatypes).

The technology of this mixed compilation and interpretation has been covered in another paper [10]. Here is suffices to note the key concept. All procedure calls jump out of the interpreter to execute a native code procedure stub. This stub performs the usual entry prolog, copying value arrays, updating the display and so on, and then launches the *Dcode* interpreter. The consequences of this design are that the interpreter does not need to know whether the called procedure is interpreted or compiled, since the calling mechanism is identical. Furthermore, for programs which spend most of their time in the code of the compiled library modules, the speed penalty for interpretation is small.

Notice that with this design the interpreter indirectly is recursively activated. Interpreter and application may alternate their activation records on the runtime stack.

1.4 Parameterisation of the stack machine

During a short sabbatical visit to Tübingen by the author, a major cleanup of the **gpm** frontend was completed. After this rewrite, all implementations of **gpm** shared identical source code. That is still the situation today, despite the fact that the code runs on versions with different byte packing order, 32-bit or 64-bit word size, and with many different parameter passing conventions. It is worthwhile seeing how this is achieved.

Firstly, it should be noted that the situation illustrated in Figure 3 is somewhat of an ideal. Truly architecture-neutral forms, such as OSF’s *ANDF* and the bytecodes of the *Java* virtual machine[20] obtain their architecture neutrality by passing huge amounts of symbolic information along with their instructions. *Dcode* is intended to be a much lower-level form, and we wished for such things as offsets into data areas to be computed by the frontend.

In the final design, the **gpm** frontend is parameterised at runtime, by reading a small “configuration” file. The file specifies such things as the data alignment constraints, the byte packing order (endian-ness) of the target, and some technical details of the parameter passing mechanism. All of these specifications apply to the target machine, rather than to the host machine which is executing the frontend. Thus if **gpm** is running on an *INTEGER = 64bits*, little-endian *Alpha* machine, and is given the configuration file for the *INTEGER = 32bits*, big-endian *SPARC-Solaris* machine, then it will produce identical *Dcode* to a native *SPARC* version.

The frontend needs to know the target data alignment constraints, so that it can correctly compute offsets of fields on the target machine. On some machines
the alignment rules are different for structures and for parameters, so that more than one alignment characteristic may need to be specified.

The problem of endian-ness is a difficult one, and became significantly more difficult with the introduction of value constructors in ISO Modula-2. Consider the following declarative code fragment

\[\text{CONST } \text{Foo } = \text{Table}\{ \text{expression list} \};\]
\[\text{Bar } = \text{Foo}[25] + 3.0;\]

Here is the problem: an image of the constant Foo needs to be constructed in the frontend, and output in correct byte-order in the output file. This argues for constructing the image using the representation conventions of the target architecture, so that a simple byte-by-byte dump of the image can be performed. Unfortunately, the second line requires us to extract a component of the constant at compile time, in the representation conventions of the host architecture. The key issue here is not whether host and target are big- or little-endian, but whether they are same-endian or cross-endian. In the event, gpm constructs such values using target conventions. The endian-ness of the target is specified by the configuration file, and the compiler tests the endian-ness of its host during startup. Thus, for the above example the element extraction code of the frontend will be selected, “on the fly”, according to the value of the initialisation crossEndian Boolean.

The parameter passing conventions are important to the frontend, since there are different constraints on parameter passing order for machines which push values on the stack, or pass values in machine registers. Stack discipline obviously demands a fixed order of “pushing”, while machines which pass parameters in registers must serialise all calls. What this means, in the second case, is that all parameters which require function calls must be evaluated before any parameters are moved to the parameter assembly area (a set of designated registers plus the calling stack for additional parameters).

The passing of records and arrays by value is even more variable. Figure 4 shows three ways by which such parameters might be passed. In this diagram, the stack grows downward on the page, parameters are above fp, while local variables are between fp and sp. For value arrays, conformance with C is not an issue, so all versions of gpm currently use the method of diagram (a) for open arrays. Most architectures correspond to diagram (b) for value records, but SPARC conventions correspond to the method in diagram (c). Finally, for fixed arrays in ISO Modula-2, fixed-length arrays must be marshalled by the caller, since element-by-element value conversion and range checking may be required. Thus the method of diagram (c) is used for all such array parameters on all targets.

\[1\] Alternative approaches, such as dumping a symbolic representation of the structured constant, were rejected, since they would have required the backends to understand the semantics of value construction.
Fig. 4. Copying structured value parameters
(a) Passed by reference, copied by called procedure
(b) Copied by caller, passed by value
(c) Copied by caller, passed by reference

2 The Gardens Point Architecture

The overall architectural concept of the Gardens Point compilers was already shown in Figure 3. In principle, the instructions of the abstract machine might be passed from frontend to backend in a text file, a binary “byte-code” file, or via a procedural interface (but see Section 2.3 for a limitation). Figure 5 shows a Decode interface with and without an intermediate file. The procedural interface into the Decode-writer and out of the Decode-parser are essentially the same.

The Modula-2 frontends are well known, but we also use our backends on Decode produced by frontends for other languages. A modified version of Fraser and Hansen’s kc [9] emits Decode to create a rather nice ANSI-C compiler. Other languages which have been used are Oberon-2, Sather 0.5[21], and a number of experimental languages including various levels of CLANG[24], and a very minimal object-oriented Oberon-2-based language called Luna[6]. Luna is an interesting case, since despite the minimality, the frontend is written in its own language. Java[21] compilers for all supported platforms are currently under construction.

2.1 The abstract stack machine

The heart of the architecture is an abstract machine, the D-machine. The Decode form is defined by the reference document[12], which is available publicly. The
form has undergone revisions on a regular basis, with major revisions occurring when 64-bit integer support was added, when more information was made available for global optimisers, and most recently when the exception handling model was upgraded to support Java exceptions. As it turns out, exception handling in ISO Modula-2 fits comfortably into this new framework, although it does not utilise all of new capabilities.

_Dcode_ is an instruction set for an abstract stack machine, much in the style of _Pcode_ [25, 1] or _Ucode_ [22]. All computation is performed on an evaluation stack, and instructions implicitly take their operands from the stack. The semantic level of the instructions is a little lower in _Dcode_ than in most such machines. The instruction set does not, for example, include any “load” instructions. Instead, there are “push address” and “dereference” instructions of various kinds which compose to create the operations of a typical load. The point of this choice is to expose more detail to the pattern matching capabilities of the backends. As well as the evaluation stack, the _D-machine_ assumes an activation record, assumed to be placed on some runtime stack. The stack-frame abstraction assumed by _Dcode_ is illustrated in Figure 6.

Abstract stack machines tend to be distinguished by the various mechanisms used to specify parameter passing, and the way in which non-top-of-stack values may be accessed. The _D-machine_ is unusual in both respects, in an attempt to facilitate both interpretation and optimisation of its output.

The passing of parameters in the _D-machine_ relies on a two step process. In the first, the actual parameter is evaluated onto the evaluation stack. In the second, a _mkPar_ (“make-parameter”) instruction pops the top-of-stack value and moves it to an abstract parameter location. The called procedure finds these values in its own activation record, at known offsets relative to an abstract _argument pointer_. Each code-generating backend will know which locations in its activation record actually correspond to memory locations and which to register values. Interpreters implement the _mkPar_ instruction by moving the top of the
evaluation stack to the correct spot in the activation record. This particular design allows more freedom in the order of evaluation of actual parameters, since evaluated parameters may be stored on the stack.

_Dcode_ does not allow access to values below the top of stack. Instead it provides a non-destructive “copy-to-temporary” instruction. These temporaries are assigned locations in the activation record. The _Dcode_ definition guarantees that these locations will only be accessed via a corresponding “push-temporary” instruction, so that simple code-generators may, without further analysis, move such values to machine registers.

It is interesting that the _Java_-virtual-machine (JVM) makes rather different choices. The _JVM_ passes arguments on the evaluation stack, thus pinning the computation order\(^2\). Similarly, the _JVM_ has a number of instructions which access values at different depths below the top of stack. Of course, since _JVM_ has no concept of offsets in the current activation record, the design choices were much more constrained.

The design of abstract machine instruction sets has two conflicting goals. The number of instruction may be made very small, but then many instructions will need additional “mode” information. This extreme is well suited to procedural interfaces in which each instruction is a separate procedure, and the mode values just become parameter values to the call. At the other extreme the instructions might have no modes at all, with separate instructions used for (say) “add”, “add with signed overflow detection”, and “add with unsigned overflow detection”. Provided that the number of instructions does not exceed 255, such a choice would lead to the simplest and fastest interpreted implementations, with the densest code.

---

\(^2\) This fixing of the order is not an issue for _Java_, since the language definition demands fixed evaluation order in any case.
The D-machine lies intermediate between these two extremes, with a total instruction count of 134, but with many instructions still having mode information. At first sight, this number of instructions may seem rather high. Nevertheless, it must be remembered that the form provides facilities for achieving proper semantics for almost all kinds of languages. Deco codes supports ten different storage data types, ranging from signed and unsigned bytes through to 64-bit integers. Whole-number arithmetic is supported at two precisions: word-sized and 64-bit, and two IEEE floating point types are supported.

In order to support various languages, arithmetic with and without overflow trapping is provided, together with multiple division and remainder operations. There are no less than 16 floating point conversion operations, 9 of which have both trapping and non-trapping versions.

If Deco were being designed now, the benefit of hindsight would see the number of instructions sharply reduced, with more mode information attached to each instruction. This would trade speed for space in pattern matching code generators, and would not restrict interpreters, since it is a simple matter to expand the canonical form in a bytecode writer module.

2.2 The runtime support system

Almost all compiler systems depend on the presence of some kind of library support at runtime. For example, most Modula-2 systems would implement intersection and union for large multi-word sets by means of a library function, rather than by generating inline code. Other languages require different language primitives to be supported by the runtime system.

It is a design goal that as far as possible frontends should be target independent, and backends should be language independent. With Deco this goal is approached by parameterizing frontends for those target attributes which are unavoidable. Backends should be entirely language independent, at least for conventional procedural languages.

Both frontends and backends require knowledge of the facilities of the runtime support system. This poses a challenge, since most languages require language-specific runtime support, particularly if they attempt to provide error messages which are presented in terms of the source language semantics. Unless care is taken, knowledge of language specific runtime support diffuses into backends, defeating the objective of language independence.

In the Gardens Point compilers this challenge is met by separating the runtime support for any particular language into two parts. One part is language independent, and includes such things as range and index bounds checking, and the trap handler for hardware traps. This part is conventionally named gprts. The other part is language dependent, and contains such support procedures and traps as are specific to the particular source language. These modules are conventionally named gp.Xrts, where X denotes the particular frontend.

Backends are permitted to be dependent on the facilities of the generic runtime support system, but should never have knowledge of the language specific facilities. This implies that whenever a frontend wishes object code to reference
a particular facility of the language dependent part of the runtime support, then the name of the facility must be passed as an explicit name in the Decode.

Consider the occurrence of a runtime range test in a program. A range test instruction "test" is part of Decode, and each backend knows how to best implement it on that target. Frontends specify the name of the trap which should be called if the test fails, so as to distinguish between index and range test traps for example. Typical Decode for a test against the range \([-5,7]\) would be

\[
\text{test } \_\_\_\_\text{gp\_rTrpLHI}, -5, +7
\]

... In the event that the test fails, the generic runtime will produce an appropriate error message.

Now consider the situation with a primitive specific to one language, but which requires runtime support. The string LENGTH() function of ISO Modula-2 is just such an example. Because this is specific to one language the support is in gpnrts. The Modula-2 frontend knows about the interface to this function, but backends cannot know about it. The function is invoked in Decode by

\[
\begin{align*}
&\text{<evaluate address expression as first parameter>} \\
&\text{<evaluate array or string HIGH as second parameter>} \\
&\text{call } \_\_\_\_\text{strLen}, 2 \\
&\ldots
\end{align*}
\]

Of course, the implementation of this function may be quite different for different targets, but frontends need only to understand the calling convention.

The rules for enforcing the abstraction principles are as follows

facilities required by "all" languages are in the generic runtime and the names of the entry points are allowed to be known by frontends (see the use of \_\_\_\_\text{gpRTrpLHI} above as an example)

facilities required by "all" languages, but only on a particular target are in the generic runtime for that target. The names of the entry points must not be known to any frontend (machines which require a table of unit vectors to efficiently implement the Decode setIn instruction are an example)

facilities required by a single language are implemented in a target-specific manner in the runtime module for that language. The frontend must explicitly generate calls to these facilities in the Decode. Backends must not know of these facilities (support for large set intersections in Modula-2 is an example)

This idea of separation of runtime support into generic and language specific modules has worked extremely well. In practice, the construction of mixed language programs which link the generic module and several of the language-specific support modules presents no problems, provided that an appropriate naming discipline has been enforced.
2.3 Interface bloat

One of the arguments against abstract machines as intermediate representations is that they create too narrow an interface. William Waite has compared such interfaces to a funnel through which all information must be forced to get from frontend to backend.

Against that view must be weighed the wide use of such forms in production compilers. Instructions for a stack machine are easy to generate, and provide an easy environment with which to experiment. For example, in the testing of our backends we have often used handwritten Decode sequences. Typically we wanted to stress the backend with a pathological but legal Decode sequence, but could not easily work out how to make any of our frontends generate such a sequence.

Nevertheless, in the last few years, we have started to suffer from what we call interface bloat. Each experimental optimisation which we wanted to apply required, or at least “wanted” just one more attribute from the frontend. In some cases it is possible to avoid the addition of attributes by the recomputation of information in the backend, but in other cases we have simply added more “noise” to the interface. Some of the optimisations in which we are currently interested require even more information from the frontends, to perform strong alias analysis, for example.

It would seem that much of the problem might be avoided if a procedural interface is provided between frontends and backends3. In this case, the procedure calls from the frontend to the backend implement the “instructions” of the bare D-machine, with optional attribute information obtained by the backend through callbacks to frontend enquiry functions. This is an option which we will probably implement, but it does not work for all languages for all targets.

Some machine code conventions require the visibility semantics of static locations to be known at the time of code selection in the backend. In effect, if a reference to a static variable occurs, then it must be known whether that object is exported or not. Of course this presents no problem for Modula-2 or Oberon-2, but is hard to guarantee in language C. In C, variables or functions may be declared extern just as a trick to provide a forward reference. Until the last line of a C source file has been read, it cannot be known for sure whether such names are really imported, really exported, or just static and local. Language C frontends which do not buffer the whole of a compilation unit before starting to emit output, simply cannot resolve this problem. Of course a two-pass backend would also solve the problem. In our experimental, lcc-based C compiler we have adopted the sneaky policy of creating two output streams from the frontend. One has the declaration information, and is produced only after the last line of source has been read. The other is the Decode instructions which are emitted incrementally as each function of the source is processed. At the start of this file we insert a file inclusion command

```
#include <the declarations>
```

It seems therefore that the C compiler is stuck with a file interface, unless a new

3 Note that in this case the frontend and backend really become a single program.
frontend is produced which either buffers a whole compilation unit, or makes two passes over source files.

3 The Various Backends

Details of a typical frontend have been given in the introduction. In this section the design of the backends is described.

3.1 Architecture, commonalities and differences

All of the Gardens Point backends have the same overall structure, as shown in Figure 7. A parser, parses the Dcode emitted by the frontend, and an explicit

control flow graph (CFG) is created. The instructions of the Dcode are passed to a code selector. The code selector produces virtual assembly language (VAL), which is placed in a code buffer. Only one procedure is processed at a time, so that only one code buffer is used at any time.

The virtual assembly language form is the assembler for an idealised version of the target machine. Thus an indefinitely large number of registers are assumed to be available, with each temporary value being developed into a new register. We have also added instructions which are “missing” from the target instruction set, when the immediate expansion to multiple instructions would not uncover significant, additional optimisation opportunities. For example, for the

Fig. 7. Architecture of the backends
Intel iapx86 architecture, trapping arithmetic instructions are added. These instructions expand out later to the corresponding, available instruction, followed by a condition code test-and-branch instruction. Dealing with this pair as a single instruction significantly simplifies the backend, since if the virtual instruction is moved or deleted, then the test moves or disappears automatically.

On all 32-bit platforms 64-bit arithmetic instructions are added. These instructions are expanded, either inline (for addition and subtraction) or into a function call (for multiplication, division and remainder). after global optimisation, but before register allocation.

The VAL of each target is different, so that the code of all modules after the parser is unique to each target architecture. Of course, the code of the corresponding modules for each target is similar in structure, and differs according to the detailed semantics of each instruction in the target instruction set. This lack of commonality is an irritation, since the maintenance of so many different modules is difficult, particularly as some of the code depends on subtle features of each instruction semantic. One of our current research projects is to take a generalised description of the module, together with a declarative specification of each target instruction set, and automatically produce these modules. We consider this to be a more promising approach than, for example, the expansion of the internal form out to a register-transfer-list form.

The final, filewriter module emits assembly language for most targets, although we have produced object files directly also. The reason for this choice is simply one of convenience. For example, most of the Intel iapx86 backends produce almost identical assembly language as output, with all details of the various object code formats hidden inside each assembler. Nevertheless, we are considering producing object code directly for some selected targets. There is particular benefit in going directly to object code for those targets for which the available assemblers do not process high-level language debugging information.

### 3.2 Shadow stack automata

Until 1995, all of our backends performed code selection using interpretative state machines, which emulate the abstract machine stack state at compile time. Such code generators are called shadow stack automata.

Despite their widespread use in production compilers, shadow stack automata have almost no written literature, with the design of the data types for the compile-time stack being a black art passed on from master to apprentice, and refined through a process of trial and error.

For the simplest possible RISC machines, the design of the automaton can be easily understood. During code selection, the shadow stack emulates the stack of the abstract machine. Thus values are pushed, operated on and popped from the shadow stack in response to the commands of the Decode. Each entry in the shadow stack has a tag, which specifies in what form the value will be at runtime. Possible values are “literal”, “value in register-N”, “value in memory at address A”. Whenever an operation is performed, the operands must be transformed into a state for which the required operation is defined, emitting one or more VAL
instructions if necessary. Once a more complicated target machine is required, the design becomes much more murky. For example, suppose an \textit{add Decode} instruction is encountered, when the two top elements on the shadow stack are in registers. On a simple machine, the response would be to allocate a destination register, emit an add instruction to the \textit{VAL} buffer, and replace the two top-of-stack values by the destination register value. Even on \textit{SPARC} this would be the wrong response. On such machines, the emission of an add instruction should be deferred, since the add might be folded into a two-register address mode of a later instruction. Instead, a new type of symbolic entry must be placed on the shadow stack, encoding the fact that this value may be realised later (if necessary) by adding together registers $N$ and $M$.

It is an irony that the design of shadow stack automata can now finally be understood in terms of the non-terminal forms of the bottom-up tree rewriting code selectors that will surely replace them.

3.3 Bottom up rewriting

It is possible for shadow stack automata to generate extremely high quality code, but the quality comes at a price. Every "new" special pattern which is inserted into the code selector requires some localised code in the pattern recognition for those \textit{Decode} instructions which use the new pattern. Moreover, if a new pattern requires the invention of a new symbolic type for the shadow stack, then the changes are spread out over the patterns of \textit{every} instruction.

Bottom up tree rewriting is currently the technology of choice for code selection\cite{16}. Such code selectors use dynamic programming to select code which is locally optimal with respect to given instruction costs. Furthermore, such code selectors may be generated from declarative specifications of the tree rewriting grammar.

After some experimentation, we have begun changing over all of our code selection to this technology. We are using the tool \textit{MBURG}\cite{11,14} which produces rewriters in \textit{Modula-2} from specifications of tree grammars, which are annotated with associated semantic actions.

Early experience shows that \textit{MBURG}-generated code-selectors take up approximately 50\% more space than our shadow stack automata, and run more slowly by a few percent. The advantage of the change is that better output code is generated, and the code selectors are much more maintainable. In the case of the \textit{SPARC} architecture, there appears to be an additional speed penalty, which we hypothesise results from the register-window spilling caused by the multiple deep recursions which these algorithms typically produce.

Although this technology is usually applied to tree-grammars which represent the instruction set and address modes of the target, much more is possible. Tree rewriters are able to recognise patterns which span several machine instructions. We have incorporated patterns into our grammars which arise from source language idioms, and make surprising savings.

It may seem ironic that current compilers walk over a well-formed tree in the frontend in order to produce a kind of flattened postfix representation which
is then reconstructed into a tree in the back end. However, the two trees are not isomorphic, since trees in the front ends are language-specific, while the code trees in the back ends are language-independent. Consider the MOD operation for integer operands in Modula-2 and Modula-3. The same syntax in the two source languages would almost certainly be represented in the same way in the two front ends. However, as shown in Figure 8, the Decode sequences are different, due to the different semantics of MOD in the two languages. Note here that

\begin{verbatim}
Modula-3
...
<push left operand>
<push right operand>
mod intOver
...

Modula-2
...
<push left operand>
<push right operand>
dup1
test __sp_ModTrp,O,MaxInt
mod intOver
...
\end{verbatim}

**Fig. 8. Decode for MOD in Modula-3 and Modula-2**

the Decode instruction “mod TrapMode” implements the general, four-quadrant modulus operation directly used by the Modula-3 code. The Modula-2 version duplicates the top-of-stack value holding the right operand, and then performs an explicit range check on the copy in order to exclude the half-plane over which MOD is not defined in Modula-2.

Thus the idea of creating a treeRewriter which operates on an AST is doomed to failure, if the same tree grammar is to be used for rewriting trees arising from multiple languages. Instead, the trees-walker for each separate language frontend understands the transformation of its own AST form to the language independent Decode form, or equivalently, the reconstructed code tree form.

### 3.4 Register allocation algorithms

The very first Decode backends used a very fast bin-packing register allocation algorithm which worked very well, but placed constraints on the form of the control flow graph as it appeared in the Decode. In particular, only forward control jumps were permitted, except for the back-edges of well-formed loops, which had to be explicitly marked in the code.

This constraint was trivial to satisfy for languages with well structured loops, but proved to be impossible to satisfy with C. Since we were in any case interested in more powerful methods of global register allocation, we introduced graph coloring register allocators for all versions. These allocators find, for each computed value, the exact set of connected program locations at which the value is live. This set is the live-range of the value. The overlapping of live-ranges defines an
“interference graph” the minimal coloring of which is isomorphic to the optimal register allocation problem.

For general graphs the exact solution of such graph coloring problems is known to be infeasible, but many excellent heuristic solutions are known [5, 4, 15]. Such register allocators sometimes use memory in a profligate fashion during compilation, but give excellent results.

As with much else in the backends, the designs of the register allocator modules in each version are essentially the same, but contain numbers of subtle differences which depend on the detailed semantics of the instruction set. A current research project seeks to automatically generate graph coloring register allocators from declarative machine specifications. The most challenging aspect of this work is to design the heuristics which must be used when the register supply is exhausted and values must be spilled to memory.

3.5 Global optimisations

Global register allocation algorithms such as the graph coloring methods perform register allocation for a whole procedure at a time. In order to do this, it is necessary to solve a dataflow problem on the $CFG$ of the procedure.

Having gone to this effort for register allocation, it is a small step to provide a dataflow analysis framework for more general global optimisations. All of our backends perform global optimisations based on a program representation form called static single assignment form [7, 3].

All backends provide for global common subexpression elimination, constant propagation and value tracking. The effectiveness of these algorithms depends on the richness of the alias information which is passed from frontend to backend.

Placing the optimiser after code selection so that it operates on $VAL$, as shown in Figure ??, is an somewhat unusual choice. Indeed one of the original advantages claimed for the Ucode form, was that it allowed machine independent optimisations to be performed. Some production compilers, such as Hewlett-Packard’s HPPA compilers, make the same choice as we do, but these examples are usually targeted on a single architecture. The argument in favour of placing the optimiser late, is that machine idioms are exposed to the optimiser in a way that would not occur if the optimisation was to be performed on the $Dcode$. Against that advantage must be weighed the fact that every backend must have its own, distinct version of the optimiser. In our case, the code which transforms the $VAL$ into static single assignment form is almost identical from version to version, but is sprinkled with minor differences arising from semantic subtleties of the target instruction set. Part of our continuing research is to find ways of parameterising these semantic variations, so that the optimisers can be generated automatically from machine specifications.

$Dcode$ still defines marker symbols which are used by frontends to signal the beginning and end of structured loops. Since it is not possible to guarantee the accuracy of such information for languages with gotos, backends must now find all loops without reference to the hints from the frontend. This is an absolute
necessity for all of those backends which perform loop optimisations. Within the
next revision cycle all backends will include these loop optimisations.

Backends abandon all attempts to perform global optimisations on any pro-
cedure that is found to have the kind of irreducible flowgraph which arises from
careless use of unstructured jumps in the source code. This avoids most problems,
but the global register allocator must still be able to perform a correct register
allocation in such cases. There are known algorithms for transforming irreducible
flowgraphs so that they become tractable, but it is against our philosophy to go
to extraordinary lengths to remove the result of programmer perversity.

4 Implementation Languages

For the most part, the frontends are all implemented in the language which
they compile. An exception is Diane Corney’s gpm and is is arguable that this
compiler would have benefitted from the challenge of bootstrapping itself at an
early stage.

All of the backends are implemented in the Gardens Point dialect of Modula-2.
With a small number of differences, this is close to ISO Modula-2. In particular,
our code makes frequent use of value constructors, both statically and dynam/-
ically assembled. We find the array constructor types very useful for mapping
tables, with record constructors less so. However, we have stumbled into error on
a number of occasions by using these constructors, and believe that their design
is intrinsically flawed.

Consider a typical constructor which creates an array of values indexed on
some enumeration type. Such a table might be useful, say, for mapping an enu-
meration of primitive datatypes to an opcode value

```plaintext
TYPE CodeTable = ARRAY DataType OF OpCode;
CONST loadCode = CodeTable{lbu,lb,lhu,lh,lw, ... };
...
Emit(loadCode[type], ... );
```

where the opcodes mean “load byte unsigned”, “load (signed) byte”, and so on.

The problem with this, is that there is no inherent ordering for such enumera/-
tions. There is thus no logical reason that the order of values in the enumeration
might not be changed. However, if the order ever is changed, then the constructor
code will become incorrect in a way that no compiler can detect. We believe that
this danger is so insidious that we have largely abandoned the use of constructors
for tables indexed on enumeration types.

It seems that if constructors are to be both useful and safe, then a syntax
extension allowing named associations of index values and component values is
required, rather than the current positional association.

4.1 Minor extensions

gpm currently implements the following minor language extensions.
**Executable assertions:** From the start, gpm compilers have supported a low-cost executable assert function. This is a very simple facility, which produces an inline Boolean test in such a way that non-trapping code takes no branches. There are two versions, one which automatically produces a module-name and line-number message, and another which allows the program to specify an explicit message string. We use both forms prolifically within the compilers.

It is hard to think of any more useful single addition to any programming language. The use of these assertions has paid off time and again during the ten-year history of the compilers. Typically, some experimental modification of one of the components has violated an intrinsic assumption of some other algorithm, and the failed assertion has given early diagnosis of this.

The Assert function must be explicitly imported from a system module, although we believe that it really should be pervasive, as is indeed the case for Oberon-2.

**Additional conversion operations:** gpm provides additional real to whole-number conversion modes, which allow for rounding to nearest (ROUND), and rounding to minus infinity (ENTIER), as well as the standard round to zero functions (INT and TRUNC). These non-standard functions are hidden in module SYSTEM, and are invisible unless implicitly imported.

**Additional datatypes:** gpm has always made the guarantee that the size of integers is at least 32-bits, and that the default real type should be IEEE double precision. We have provided for lower precision real types by means of a non-standard type SHORTREAL. For all current versions, LONGREAL is the same as the default real type, but in future some versions might provide extended double types, or the increasingly popular 128-bit reals.

Since our backends must be able to manipulate data of all the primitive types required by our various frontends, we decided that it was necessary to support the 64-bit integer type. We were reluctant to call this long type LONGINT since this name is already used by many thousands of our users programs which have been ported from 16-bit dialects to gpm.

We call our 64-bit type HUGEINT. It is a whole-number type, but is not classified as an ordinal type. Thus, variables of this type cannot be used as array indices or for-loop indices. The type has a full set of arithmetic operations, including both kinds of divisions and remainders. In keeping with the style of Modula-2, there are no implicit coercions to or from this type, but there are named conversions to and from the signed and unsigned type, and a full set of conversions to and from the real types with the usual choice of three different rounding modes.

All of the facilities of huge integers are invisible unless imported from the system module “HugeInts”.
4.2 Extensible arrays

One of the very widely used data structures in the backends is the dynamically allocated, variably sized array. These are used for sets of virtual registers, interference graphs, code buffers and various list types.

In an experimental modification to gpm, we have added a new type constructor which provides for extensible arrays. These arrays automatically expand to accommodate append operations beyond their original capacity. A short technical report describes this extension in detail [13].

These string types, as we call them, can be declared for arbitrary element types. In particular, if the element type is another string type then a multidimensional, extensible array is created.

Accesses to elements of these string types uses the ordinary array index syntax, and are always index bounds-checked against the current length of the string, rather than the length of the container in which the string is embedded.

Figure 9 represents the runtime implementation of a string type. Each variable has a descriptor block, which records the current high limit of the string, as well as the number of elements of the container in which the string currently resides. If a string needs to expand beyond its current container size, a new container is allocated, and all of the current string copied across. With our current allocation strategy, strings implement lists and stacks using amortised doubling.

We are unsure whether the introduction of a new type constructor can be justified for general programming. It is true that the same effect could be obtained using an abstract data type, but each different element type would require a separate module to support it. In our case, given the ubiquity of extensible arrays in our code, we are finding the extension to be both efficient and convenient.

5 Current Projects

A number of research projects are based around the infrastructure of the Gardens Point compilers. The most ambitious of these is "Gardens" [23], a network of workstations project which involves both language and system design aspects. Associated projects are researching some aspects of garbage collection and of compiler-assisted heterogeneous task migration.
As as well as the ongoing work on code optimisation, a current project seeks to create tools for generating backend modules from specifications. Initially this project considers register allocators and some dataflow optimisations.

6 Conclusions

This paper has given an overview of a project which is now 10-years in the making. The original aims have been somewhat expanded, so that now the project supplies a framework within which a broad variety of research topics may be pursued, and a number of tools which are used every day by many people worldwide.

Many of the technical decisions which have been made turned out to be sound, others less so. There are a number of decisions which, given the benefit of hindsight would now have been taken differently. Some of these have been hinted at here.

To some degree compiler writing is at an interesting crossroad. For the computer vendors, there is an increasing emphasis on finding very specialised optimisations which apply to their own machines, or even to just specific configurations of their machine. Many of these optimisations require whole-of-program analysis, and involve such things as improving cache behaviour.

The other path is towards new compiler technologies which can be applied to "just in time" compiling techniques, and are applicable to extensible and dynamically configured applications.

While the Gardens Point project still has interests in optimisation algorithms, it seems that the second path is the more fruitful one, and the one which leads to the more interesting research opportunities.

6.1 Acknowledgements

The Alpha version of gpm was produced with the help of a Digital Equipment Corporation "Alpha Innovators Grant". The process of moving our code to our first 64-bit target is described in some detail in a report[19].

The current work on Java frontends is a project of the Distributed Systems Technology Centre, a centre set up under the Australian Government’s Cooperative Research Centre program.

Most of the papers and technical reports about the project referenced here are also available from the URL http://www.dstc.qut.edu.au/~gough/

References


This article was processed using the \LaTeX macro package with LLNCS style