Efficient Code Generation for a Domain Specific Language*

Andrew Moss and Henk Muller

Department Of Computer Science
University Of Bristol

Abstract. We present a domain-specific-language (DSL) for writing instances of a class of filter programs. The values in the language are symbolic and independent of a concrete precision. Efficient code generation is required to fit the program onto a target device limited in both memory and processing power. We construct an interpreter for the DSL in a language specific to the device which contains the semantics of the target instruction set embedded within a declarative meta-language. The compiler is automatically generated from the interpreter through specialisation. This extension of the instruction set allows the construction of an interpreter for the DSL that is both simple and clear. In particular it allows us to declare static representations of the symbolic values, and have the specialisation of the code produce operate upon these values in the instruction set of the target device.

1 Introduction

Our target domain is the sensor architecture of a wearable computer. This domain has tight constraints on the energy usage of system components; requiring the use of micro-controllers which are power efficient. Such devices have a limited instruction set, little data memory, and a small store for program code. The current method for programming such devices is to write code directly in assembly language. A C compiler exists, but the size of the code produced makes it impractical for real programs.

The class of programs that execute in this domain are hard real-time processes. They input the raw sensor readings, which are filtered to identify features for further processing by the application on the main processor. The sensor is responsible for polling the data, filtering it, and communicating the result over a bus to the main processor.

From the designer's viewpoint, the filter is a set of equations defining the relations between input and output. The equations are composed of simple arithmetic operations but their semantics are defined over arbitrary precision bit-strings. This abstract viewpoint is a long way from the actual implementation as a series of instructions.

* This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 ASAP project

Our goal is to construct a DSL for the designer that matches their abstract view as closely as possible. This abstract program compiles into a concrete executable for the target device. In order to compile the code, the arbitrary precision values must be refined into appropriate approximations using static precision. It is important that the efficiency of this generated code should match that of an assembly language programmer — otherwise the generated program will fail to fit upon the device.

To meet this compilation goal the equational system specified in the DSL must be converted into a sequence of device instructions. The equations of the DSL contain no explicit control flow; there is a single implicit loop around the set of equations in the program. This loop forms an infinite loop around the statements in the generated program; each filter is designed to execute for as long as there is power to the sensor. Within the loop there are a series of expressions that compute the value of each of the variables in the filter. Each iteration of the main loop is a single input/output cycle on the sensor and can be scheduled between the hard temporal constraints.

Our compilation technique is to write an interpreter for the DSL, and automatically generate a compiler from that interpreter through specialisation. The novel aspect of the technique is that the interpreter cannot be constructed directly in the target language. Instead the interpreter is constructed in two parts; an interpreter of the target instruction set is embedded within a declarative meta-language to form an extension of the target language, and an interpreter of the DSL is written in this extension. The extension consists of operations in the meta-language and calls to the target interpreter.

In order to compile a DSL program into the target language, the DSL interpreter is specialised with respect to the program. The specialiser operates on the meta-language, and has no direct knowledge of either the DSL or the target language. Specialisation of the DSL interpreter produces a result in which the calls to the target interpreter are left in the residual. These calls to the target interpreter are *syntactically isomorphic* to the target instruction set. By careful construction of the DSL interpreter we produce a residual program that is written in the meta-language but only contains calls to the target interpreter. Thus the resultant code is syntactically isomorphic with the target language and can be converted by a simple syntax conversion. This process compiles code from the DSL to the target language using a specialiser that performs source-to-source transformations *only* within the meta-language.

2 Target device

Our target device is a PIC-16F84[1] micro-controller. We have chosen this device as it is representative of many real-world applications in the robotics and pervasive computing fields. The main attractions of the device are low cost (under a dollar) and low power consumption (several milli-watts). Unfortunately this simple design creates limited resources and an awkward programming model.

Each micro-controller uses 8-bit logic and arithmetic. Operations are performed in a nominated working register, the other registers are devoted to status and control flags and a scratch area for the programmer. As external memory is not connected to these devices, the scratch area (about 60 8-bit locations) forms the only memory available for the programmer to use[1]. The program is limited by the size of the program memory to about 1000 instructions.

The program and data memories are not shared, and it is not possible for the device to modify its program memory while it is running. This limits the possibilities for generative or self-modifying code but it does make analysis of PIC programs easier.

The instruction set of the PIC supports basic arithmetic operations (addition and subtraction) but more complex operations (eg multiplication and division) must be written by the programmer. There is support for logical AND, and both inclusive and exclusive OR. Control flow is limited, with decisions being performed by conditional *skip* instructions. These can either execute the next instruction, or skip over it, depending on the value of a bit in the register file. Logical shifts are not supported, although rotation can be performed through a register and the carry bit, a single bit at a time. Logical shifts can be constructed from combinations of rotate sequences and masking.

The PIC uses a normal model of control-flow; each instruction in the program has an integer label and a register called the PC selects the current instruction. Execution of each instruction affects the PC and so directs the flow of control through the program. The important point is that the target machine is imperative and the active program point is part of the state being passed through the computation. This differs from the control flow in our meta-language and we shall explain how we have embedded this imperative state machine in a logic language in Section 5.

The code density of PIC programs is low because of the decisions made to select this instruction set. Logical shifts and rotations of multiple bits take several instructions which are costly both in processor cycles, and slots in the limited program space. Each conditional split in the control flow must be constructed from multiple instructions. Implementing multiple precision arithmetic is costly as we must extract bit-string from different locations and merge them in order to perform operations. This requires both shifting of data within registers and conditional code to interpret values at different precisions. However, as we will show this implementation can be achieved by specialising away the conditional code flows and choosing efficient combinations of operations. Implementing arbitrary precision (that is dynamically changing precision) would not be feasible because this approach could not reduce the code synthesised for the PIC.

[1] There is also an EEPROM but it requires several cycles and instructions to access, and has a limited lifespan

3 Approach

We are constructing an interpreter of a domain language (D). This interpreter can then be partially evaluated with a specific program in that language. The evaluation compiles the program into the language that the interpreter is written in. If we could construct this interpreter directly in our target language (T), then our compiler would be complete as shown in Equation 1 (the first Futamura projection). However, T is not rich enough to support such an interpreter, as argued in Section 2, and thus int_T^D is not directly constructible.

$$\forall i : [\![[\![spec]\!](int_T^D, P_D)]\!](i) = [\![P_T]\!](i) \tag{1}$$

If we assume that do have int_T^D available, and that we have a rich meta-language (M) that we use as an intermediate, then we can write a second interpreter (int_M^T) that can be composed with the first. So int_M^T executes the instructions of int_T^D which executes the target program. Partial evaluation of the first interpreter with respect to the second, will produce an interpreter of D that is written in M. The structure of this interpreter will retain the semantics of the target language (T), as parts of the predicates that model each instruction will be left in the residual code. This process is shown in Equation 2 where we term the language of the final interpreter M', as it contains the embedding of the semantics of the instructions in T. Programs in M' have all of the expressive power of M, but the parts of them using the predicates that model T are syntactically isomorphic to programs in T.

$$[\![spec_M]\!](int_M^T, int_T^D) = int_{M'}^D \tag{2}$$

We have constructed both $int_{M'}^D$, and $int_M^{M'}$ directly. These are the DSL interpreter and the target interpreter respectively. Handwriting the DSL interpreter requires some care, as we must ensure that when it is specialised against a program all of the predicates that do not share a syntactic isomorphism with T are correctly analysed as static and removed from the residual. Ensuring that all calls to isomorphic predicates remain within the residual is easier, as we have a single dispatch point within the target interpreter that we can explicitly tag as *rescall* in the BTA. The Ciao package supports these explicit annotations.

One advantage of this manual construction is illustrated by our compilation process in Equation 3. The only external tool that we require is a specialiser for the meta-language, there is no need to write any analysis or transformation tools that natively understand either the source language (D) or the target language (T).

$$\forall i : [\![[\![spec_M]\!](int_{M'}^D, prog_D)]\!](i) = [\![prog_T]\!](i) \tag{3}$$

A predicate written in M' is *syntactically isomorphic* to a program in T iff it contains a conjunction of terms that are syntactically isomorphic to T. A term is syntactically isomorphic to a program in T iff it is a call to a predicate in the target interpreter or a call to a predicate that is itself syntactically isomorphic to T. More informally, when a program in the meta-language reduces to one that

```
input(z), output(x), state(q,a,h,p,r)
pminus = q + (a * p * a)
zpred = h * x
resid = z - zpred
k = (h * pminus) / (r + (h * pminus * h))
x = (resid * k + x)
p = (1 - (k * h)) * pminus
```

Fig. 1. Example DSL program : Kalman Filter

consists only of calls to predicates that model the instruction set of the target, then the residual program is syntactically isomorphic with a target program as we can perform a simple syntactic substitution to rewrite it as PIC assembly code.

In the remainder of this paper, we describe the construction of the languages and interpreters required to implement this approach. The DSL language (D) is described in Section 4 and the construction of its interpreter in M' is covered in Section 6. The target has been described in the preceding section, embedding the interpreter of the target language in M is detailed in Section 5.

4 Domain Specific Language — *D*

The aim of a DSL is to allow the construction of clear programs. This clarity is achieved by abstracting away details that are constant within the domain. In the case of D all necessary control-flow in the program has been removed. Each program written in D is a set of equations defined over variables. Some variables are nominated as representing the input and output streams, some as stateful variables that preserve their values between iterations of the filter, and the rest are transient. An implicit outer loop exists around the equations that loops forever.

In order to remove the control flow within the loop we use the semantics of a simulation language for variable access. The implicit outer loop separates the execution of the program into discrete cycles. Each variable in the language maintains two states, one for the current cycle, and one for the next cycle. All variable updates write to the state for the next cycle. All variable accesses read from the state for the current cycle. Between cycles the contents of the next state are copied into the current state. This bi-state for each variable removes dependencies in the program code and means that equations can be evaluated in any order without affecting the result of the computation for each cycle. An example of this type of program is shown in Figure 1.

4.1 Variables

Each variable in the DSL has arbitrary precision; it is represented as a pair of bit positions and a bit-string. The bit-string contains the value of the bits between

$$\text{bit}(n, \text{var}(u, l, bs)) = \begin{cases} s & n \geq u \\ bs[n-l] & l \leq n \leq u \\ 0 & n < l \end{cases}$$

$$\text{where} \quad \begin{array}{r} s = bs[u-l] \\ (x_0, x_1, ...)[n] = x_n \end{array}$$

Fig. 2. Definition of bit-positions within a variable

$$\begin{array}{lcl} \text{trans}(\text{var}(u, l, bs), ou, ol) & = & \text{var}(ou, ol, obs) \\ \text{where} \quad \forall n : ol \leq n \leq ou & & \\ \quad obs[n - ol] & = & \text{bit}(n, \text{var}(u, l, bs)) \end{array}$$

Fig. 3. Transformation function

the given upper and lower bit of the variable. Each item in the bit-string has the value 0 or 1, and each position n has a coefficient of 2^n except for the upper position which has a coefficient of -2^n. The radix point is between positions 0 and -1. Each value is 2s-complement, and we round towards negative infinity. This means that all of the bits in the positions above the bit-string are sign-extended from the top bit given, and all of the bits in the positions below the bit-string are zero.

This definition of the value represented by each variable means that although we only hold a finite portion of the bit-string that exactly represents the value, all of the possible bit-positions for the variable are well-defined. This definition is shown in Figure 2.

The most basic function that we can define on these values is transformation; mapping from a value represented at one precision, to the closest representation of the value at another precision. This function is not invertible, as information is lost when mapping from a higher precision to a lower precision. This lost information means the function is not injective, and will map several values onto the same representation in the lower precision.

The transformation function is shown in Figure 3. The simplicity of the function definition is a result of the bit being well defined in all positions in the original precision. This results in a lack of corner cases and thus a uniform definition.

Each variable in the target language must have a fixed precision; the DSL program *implicitly* defines a minimum value for these precisions. It is beyond the scope of this paper to describe the different techniques of assigning precisions to the intermediate variables in a computation. Broadly the different approaches can be described as analytical techniques [2] that preserve correctness, or statistical techniques that manage the propagation of error [3–5]. We assume that a solution for the set of precisions is known that correctly represents the intention of the programmer, and this set of precisions is supplied to the DSL interpreter — that the bit-positions of each variable are static data. With this assumption

Operation	Intermediate Precision
$(au, al) + (bu, bl)$	$(\ \max(au, bu) + 1, \min(al, bl)\ )$
$(au, al) - (bu, bl)$	$(\ \max(au, bu) + 1, \min(al, bl)\ )$
$(au, al)\ \cdot\ (bu, bl)$	$(\ au + bu, al + bl\ )$
$(au, al)\ /\ (bu, bl)$	$(\ au - bl, al - bu\ ) \subset (\ au - bl, \inf\ )$

Fig. 4. Intermediate and output precisions

we focus on the automatic generation of a code operating in a given fixed-point form, rather than determining the value of that form.

4.2 Operations

Defining a transformation function first makes the definition of individual operations simpler. We no longer have to consider the various cases of which bits are contained in a value representation, and which are not. The set of operations that we support in the language is addition, subtraction, multiplication and division. Each operation first transforms both operands to an intermediate precision (this is the necessary precision to preserve correctness for each bit in the output), performs the operation between two values of that precision and then transforms the resultant value to an output precision. The intermediate precision may be larger than the output precision, for example in the case of addition where carry chains must be computed to determine the correct result. For each operation the intermediate precision with respect to a given pair of input precisions is shown in Figure 4. The value produced at this intermediate precision is then transformed to the precision of the target variable as the output precision.

For addition, subtraction and multiplication it is possible to statically determine this required precision from the precision of the operands alone. For a division operation this produced precision is dependent on the actual values supplied to the operation, and the safe approximation is a bit-string of infinite length as any divisor that is not a power of 2 will produce a repeating binary string as a result. We can still use the expression $bu - al$ as a safe approximation of the upper bit in the precision as we assume that we cannot divide by zero. The safe approximation is not representable so we use the precision that represents a subset of the values representable in the safe case. This is unavoidable when performing division on a machine with finite resources. The pseudo code in Figure 5 shows the process for performing each operation in the language.

The basic operations can be combined into expressions by the programmer. We assume standard associativity for each operator. Each expression is a simple tree of operations that we flatten into a sequence of basic operations. This requires the introduction of temporary variables to hold each intermediate result within the expression. The language that we have described thus far is executable and can be used by the designer to develop programs and test their correctness. Arbitrary precision arithmetic in the DSL is captured in the pseudo code by undefined precisions for the output of an operation. This operation is

```
perform(op,var(au,al,av),var(bu,bl,bv),ru,rl) =
    (iu,il) = Select from Fig.4 based on op
    ai = trans(var(au,al,av),iu,il)
    bi = trans(var(bu,bl,bv),iu,il)
    rs = perform op on ai and bi
    defined(ru,rl) ->
        return var(ru,rl,trans(var(iu,il,rs),ru,rl))
    undefined(ru,rl) ->
        ru = iu
        rl = il
        return var(ru,rl,rs)
```

Fig. 5. Pseudo code for each operation

not feasible on the target device. The implementation on the target requires ru and rl to be static values passed to the DSL interpreter. For intermediate values these precisions are constrained in the same manner as the temporary variables in Section 4.1.

5 Embedding The Target Language — $M' = M + T$

The meta-language (M) is Ciao [6] (a dialect of Prolog). Prolog is used because of the maturity of analysis and specialisation tools for the language and their state of integration into Ciao — in the form of Ciaopp [7]. The choice of language for M must be well-suited for the construction of the target interpreter. Most of the operations within the interpreter are mappings from one domain to another. These maps have a natural form as logical predicates. Prolog is both typeless and symbolic with a simple encoding of anything as data (even predicates) that allows a rapid cycle of formulating ideas and testing them as code. The most important feature in Prolog for our needs is that it features dynamic syntax. When writing code at several different language levels it is necessary to test the parts in isolation without constructing a correctly typed framework for them.

Each instruction in the target language maps to a predicate. M' is the combination of the Prolog language with calls to these predicates. Each transforms an initial state to an output state as directed by its parameters. An example is given in Figure 6. The *addwf* instruction retrieves two values from the initial memory state (one from the working register 0, and one specified in the first parameter), adds the two values, and then depending on the second parameter, either stores the result in the working register, or in the specified register. The carry and zero bit are set according to the result.

The memory state is represented by a simple list of pairs of integers, one with the location label and one with the current value bound to that label. The lookup and store operations retrieve the value from a label, and produce a new state with the label set to the given value. These operations are expressed clearly in a declarative language as shown in Figure 7.

```
addwf(S,F,D,S3) :-
    lookup(0,S,W),
    lookup(F,S,X),
    Ans is W+X,
    AnsM is Ans mod 256,
    (D=:=1 -> store(S,F,AnsM,S2)
            ; store(S,0,AnsM,S2)),
    statusBits(Ans,S2,S3).
```

Fig. 6. Sample implementation of T in M

```
mTransAd( (L,V1), L, (V1->V2), (L,V2) ).
mTransAd( (L,V), L2, (_->_), (L,V) ) :- L \== L2.
mTrans( S, L, X, S2) :- map(S,mTransAd(L,X),S2).
lookup(Ad,(_,S),V) :- mTrans(S,Ad,(V->_),_).
store((D,S),Ad,V,(D,S2)) :- mTrans(S,Ad,(_->V),S2).
```

Fig. 7. State transition predicates

These predicates map directly onto the target instruction set although they are executable in a logical language. In the PIC micro-controller each instruction is located at an address in memory and the semantics of the instruction-set specify how the state transitions affect the PC, indirectly controlling which instruction is executed next. In the target language T a program is comprised of a list of instructions. The control-flow is contained implicitly in the sequencing of the list. When T is embedded within M the control-flow through the instructions is the Prolog control-flow around the calls to the target predicates. In general this can be more complex than a simple sequence, although programs in M' that are syntactically isomorphic to T are a simple sequence of conjunctions by definition.

The interpreter of T implements the semantics of the individual instructions, rather than emulating the machine that executes them. This subtle difference allows Prolog code to be freely interleaved with calls to the target interpreter. The resulting code can be executed natively in M as the explicit threading of the state gives a well defined meaning to the program. If a partial evaluation removes any surrounding Prolog control flow decisions (reducing them to conjunctions) then the resultant code has a syntactic one-to-one mapping with a PIC assembly language program. The syntactic mapping has to relocate the program to an absolute address in the program memory, but as all control-flow in T is relative this is a trivial mapping.

One difficulty this abstract control-flow poses is how to interpret the *skip* instructions that conditionally skip or execute the next instruction. In the PIC with each instruction bound to a location the PC can be incremented to implement skipping. When a program in T is embedded in M there is no direct relationship between one instruction and the next in the control flow. This relation now depends on the Prolog code which calls the two instruction predicates. This difficulty is overcome using the higher-order features of the symbolic meta-

```
pic(Inst) :- Inst =.. [P,skipping(S),S].
pic(Inst) :- Inst =.. [P,skipping(S),_,S].
pic(Inst) :- Inst =.. [P,skipping(S),_,_,S].
pic(Inst) :- Inst =.. [_,(_,_)|_],
             call(Inst).
```

Fig. 8. State dispatcher

```
extendSign(S,(_,_,8),S).
extendSign(S,(Ad,Bit,Fill),Sout) :-
    Fill<8,
    OrMask  is 255 - (1<<Fill)+1,
    AndMask is (1<<Fill)-1,
    pic(btfss(S,Ad,Bit,S2)),
    pic(andlw(S2,AndMask,S3)),
    pic(btfsc(S3,Ad,Bit,S4)),
    pic(iorlw(S4,OrMask,Sout)).
alignByteInW((D,M),V,B,Sout) :-
    member(var(V,_,Vu,_,_,_,_),D),
    B>Vu,
    bitAddr((D,M),V,Vu,SignByte),
    aligned((D,M),V,Vu,SignBit),
    extendSign((D,M),(SignByte,SignBit,0),Sout).
```

Fig. 9. Example DSL interpreter operation

language. The entire state is encapsulated within a functor (eg $S \mapsto skip(S)$) when the skip condition holds. Then each PIC call is wrapped in a dispatcher that checks whether to execute or skip the predicate that is passed in. There is minimal change to programs written in M', and the interpreter merely requires the addition of a dispatcher as shown in Figure 8.

Implementing the target interpreter in M creates a language M' that allows a clear construction of the DSL interpreter. The clarity is achieved through interleaving declarative logic (symbol manipulation, backtracking and query solving) with calls that implement the semantics of instructions in T. The predicates implementing T must be passed ground values for instruction parameters, and must be fully deterministic. When backtracking is used to enumerate possible code generations in the DSL interpreter each execution of an instruction must generate a single transition in the state. Multiple transitions for a single instruction with ground parameters will generate spurious call traces and erroneous partial evaluations.

6 Interpreter of D in M'

Using the combination of the target interpreter and the meta-language that form M' it is now possible to write an interpreter for D. Throughout this paper we have argued that it is not possible to write such an interpreter directly in T.

The overhead of interpreting instructions, and maintaining a state for D and its mapping onto the device would consume all available resources.

Writing the interpreter in M' solves these problems. The state of the interpreter is composed of a static precision (upper and lower bit-position) for each variable, and a dynamic value (bit-string). The bit-string requires a mapping between partitions of the string and memory locations on the target device. The mapping can be defined declaratively in M' and passed to the interpreter as a static value. The only remaining dynamic values are the values of the bit-string — these must fit within the program memory for the filter to be executable.

We consider two representations of the bit-string. In a packed representation the lowest eight bits of the string occupy one location, each partition of eight bits occupies a sequential location in memory. In a modulo representation each bit n of a location maps to a position that is $n \; mod \; 8$. These two representations have a trade-off; packed variables take less memory but mod variables require less instructions to operate upon.

In M' we can declare disjoint clause bodies for each representation that map to different calls to target instructions. This method requires no run-time overhead and no extra code in the interpreter as back tracking at specialisation time will select the appropriate clause in each context.

Clarity is the result of operating directly on the same symbolic values in both interpreters. Performing an operation on bit-strings has a simple declaration as a uniform loop over a window of bits. Writing the same operation on locations which hold partitions of the bit-string is more complex as there are more cases to consider. All loops and queries within the interpreter that are dependent on the positions and sizes of bit-strings will be statically unrolled into target call sequences. Depending on the relative alignment of bits between a pair of registers, we must perform a different series of target instructions to implement the DSL operation.

In order to encode each DSL operation as a uniform loop over bits, we require an intermediate operation in the interpreter; alignment. This operation is composed of target calls and declarative logic. We show the simplest case of alignment in Figure 9. In this case we are attempting to extract a range of bit positions that are above those stored in the state, we extend the sign of the top bit in the represented value.

The alignment operation uses the variable declarations in the DSL state to find the bit-positions that are stored. It can then map these bits onto the representation in memory. There are several possible cases to be considered, each of which forms a predicate body. The requested bits may be above, below or within the stored bit-string. The representation of the value affects whether the bits are within a single location or span several locations.

Given the alignment operation, each DSL operation is constructed as a uniform loop over bit-positions. The iterations of the loop that access bits outside the string will be specialised away. The structure of a pseudo DSL operation is given in Figure 10, showing the uniform operation on 8-bit partitions of the value, regardless of the precision and representation in memory. The $x_{[u,l]}$ no-

$$o = a \cdot b \iff \forall p \in [o_l..o_u] \ : \ p - o_l \equiv 0 \pmod 8$$
$$carry_{[p+n,p+8]}, o_{[p+7,p]} = a_{[p+7,p]} \cdot b_{[p+7,p]} \cdot carry_{[p+7,p]}$$

Fig. 10. Mapping Of Intermediate Binary Operation

tation indicates the bits l to u in the variable x, which are instances of the alignment operation.

7 Application

We have given a formulation of the DSL interpreter that operates on a program, a declaration of the variable precisions and a declaration of variable representations. When this interpreter is specialised with respect to these parameters it produces a residual program in M' that we can convert to PIC assembly language.

The trade-off in choosing how to represent each variable is complex and hard to express. The trade-off is a constraint problem where the set of constraints to solve is specific to each program in the DSL. This constraint problem could be specified as a transformation of the program into a constraint language, but the formulation would be difficult.

As M' is a logic language we can use the declaration of representation to generate possible variable representations for the program. Each of these generated values can be supplied to the interpreter to automatically generate code using the representation. Each of these generated codes can be compared to find a target program that matches a given constraint (eg code size or memory usage). This example shows how the search space of the DSL interpreter can be explored to find solutions (target programs) that match criteria that are difficult to express directly as search problems.

The low code density of the PIC limits the complexity of an operation that we can demonstrate code generation upon. We will use the alignment case shown in Figure 9 and an entry point that includes an interpreter state with a single variable. The variable uses a packed representation within the PIC memory, its lowest bit is aligned with the lowest bit in a register location, and its higher bits are spanned onto the next location. This call to the alignment operation is shown in Figure 11.

The specialiser removes the static arguments to the call, which include the variable being accessed. The resultant code has eliminated the declarations that access the interpreter state and produced constant values as the parameters to the PIC calls. The variables in the resultant code are renamed to preserve sharing, which leaves the state threading intact in the produced code. The output is a straight-line code sequence composed entirely of calls to the PIC predicates, and can be syntactically transformed into the PIC program shown.

The interpreter state and all interpretive overhead has been removed from this operation, as the code does not modify the layout of variables in memory. Supplying different memory layouts, and variable representations produces the

```
:- entry alignByteInW(( [var(x,packed,6,-4,10,2,_)],S ), x, 7,
                      ( [var(x,packed,6,-4,10,2,_)],S2)).

alignByteInW(A,E) :-
    pic(btfss(A,11,1,B)),
    pic(andlw(B,0,C)),
    pic(btfsc(C,11,1,D),
    pic(iorlw(D,255,E)).

PIC program:
    BTFSS   11,1
    ANDLW   0
    BTFSC   11,1
    IORLW   255
```

Fig. 11. Specialisation of getSign predicate

appropriate access code. In larger interpreter fragments (such as multi-precision addition and multiplication) there is a more complex trade-off between the choice of representation, and the size of the residual code.

An alternative approach to implementing the DSL on the target device would be to write a library performing multi-precision operations. The contrast with our approach is dramatic, both in terms of complexity and the efficiency of the final code. The declaration of the alignment operation defines a series of condition checks and an appropriate body to execute in each specific context. This overhead is removed entirely by the specialiser.

Writing each DSL operation as a library function would require a set of cases that contain all of the used calls (eg 8-bit / 16-bit). Each case would need to be written separately, increasing the complexity of the implementation. Any unused bits in the computation, such as using an 8-bit multiplier on a pair of 6-bit values would introduce unnecessary overhead and the performance of the program would suffer. Our approach avoids this problem by using the specialiser to automatically determine which expansion is necessary in each context. A library approach would also require the precision for each variable to be passed to the routine. In contrast our approach reduces these values to constants, and they are implicitly defined when needed so they require no storage. The disadvantage is that unrolling the code in this way increases the program size.

8 Related Work

The approach of code generation by abstract operations is detailed in *delayed code generation* [8]. This method of producing object code for a smalltalk compiler uses intermediate operations between the source and target languages that contain abstract values. These values can be operated on through *deferred* operations that can eliminate intermediate steps in the object code. This approach differs from our method in that these operations are performed upon syntax

trees within the compiler. We execute our operations directly through a partial evaluation and reduce expressions based on the static analysis performed within the specialiser.

The uses of partial evaluation have been studied extensively [9, 10], and in particular the use within meta-programming to compile domain specific languages [11]. The novelty of our approach in comparison to these techniques is the construction of the domain interpreter within an extension of the target language. This extension allows us to use the expressive power of the declarative meta-language to write our interpreter clearly and simply. The elements of the target language are residualised during the specialisation, where-as the operations in the meta-language are entirely static. This produces a syntactic isomorphism between the residual program and the target language.

The connection between macro languages and multi-stage computations has been investigated in a functional setting by Ganz et al [12]. They formalise their macro language (MacroML) as a MetaML interpreter and use the features in MetaML to show that their language is type-safe (that macros cannot create type errors in the final program) and stage-safe (that macro expansion does not rely on run-time evaluations).

This work differs from our own in that we are using an untyped language as the meta-language, and rather than explicit staging constructs we are using online partial evaluation to separate the stages and perform the macro computation. Our work focuses on how to use an implicitly staged computation to perform macro operations; these are not limited to the operations in a conventional macro language as we can freely mix data values between the stages, allowing some runtime values to effect compile-time computations. Of course these runtime values are restricted by the static analysis used within the specialiser.

The approach in Sh [13] uses static meta-programming to embed a shader language in C++ templates. The shader programs are constructed at run-time by recording the operations executed in the meta-language. This reuses the bulk of the C++ compiler for parsing, grammar checking and code generation. This approach is similar to our embedding within a host compiler, although the recording of operations is performed at partial-evaluation time by the specialiser leaving no runtime overhead.

Previous work in compiling embedded languages [14, 15] has looked at embedded typed languages within an existing typed functional language. This allows the meta-language to act as a host compiler and produce code for the DSL. This differs from our approach as we are embedding a lower level untyped language that contains the semantics of the executable target domain. In particular, the construction of our interpreter is similar to the handwritten cogen of [14], in that we have deliberately written the output of a theoretical specialisation. We cover this aspect of the work in Section 3.

Herrman and Langhammer show a similar approach to our work in [16]. A DSL for image processing operations is constructed for a similar class of filter programs. Efficiency is also a concern in that domain and their system generates code from an interpreter to remove the interpretative overhead. The construction

of their interpreter uses explicit staging constructs rather than a specialisation approach.

9 Conclusions and Future Work

We have presented a language that uses a high level symbolic representation of our problem domain. This representation has been compiled into an efficient executable form for an extremely low level micro-controller with limited resources.

We have shown that embedding the target language in our choice of meta-language allows the construction of an interpreter that would not be feasible otherwise. This interpreter is both simple, and clear, and when specialised it produces a residual program that is syntactically isomorphic with with the target language. This achieves compilation from the domain language into the target language by a specialiser that only operates in, and on the meta-language.

Our next step will be to investigate the compiler on a realistic target from the domain. We will show how efficient the compilation process is, and investigate optimisation techniques on the generated code.

10 Acknowledgements

The authors offer thanks to John Gallagher for creating the term *syntactic isomorphism* and also for constructing a clear overview of the work. We would also like to thank the anonymous reviewers for their helpful comments which aided the structure and content of this paper.

References

1. Microchip. PIC16F84 data sheet. Technical Report DS35007B, Microchip Technology Inc, 2001.
2. Willems,M. and Bürsgens,V. and Meyr,H. FRIDGE: Floating-Point Programming of Fixed-Point Digital Signal Processors. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, pages 1000–1005, San Diego, Sep. 1997.
3. Ki-Il Kum, Jiyang Kang, and Wonyong Sung. AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems-II:Analog and Digital Signal Processing*, 47(9):840–848, September 2000.
4. Radim Cmar, Luc Rijnders, Patrick Schaumont, Serge Vernalde, and Ivo Bolsens. A methodology and design environment for DSP ASIC fixed-point refinement. In *DATE*, pages 271–, 1999.
5. Keding, H. and Hürtgen, F. and Willems, M. and Coors, M. Transformation of Floating-Point into Fixed-Point Algorithms by Interpolation Applying a Statistical Approach. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, Toronto, Sep. 1998.

6. M. Hermenegildo, F. Bueno, D. Cabeza, M. García de la Banda, P. López, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, 1999.
7. Manuel V. Hermenegildo, Francisco Bueno, German Puebla, and Pedro Lopez. Program analysis, debugging, and optimization using the ciao system preprocessor. In *Proceedings of the 1999 international conference on Logic programming*, pages 52–66, Cambridge, MA, USA, 1999. Massachusetts Institute of Technology.
8. Ian Piumarta. *Delayed Code Generation in a Smalltalk-80 Compiler*. PhD thesis, Department of Computer Science, University of Manchester, Oct 1992.
9. John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*, volume 1706 of *Lecture Notes in Computer Science*. Springer, 1999.
10. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
11. P. Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 134, Washington, DC, USA, 1998. IEEE Computer Society.
12. Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In *ICFP*, pages 74–85, 2001.
13. Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.
14. Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 9–27, London, UK, 2000. Springer-Verlag.
15. Morten Rhiger. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.*, 25(3):291–315, 2003.
16. T. Langhammer C. Herrmann. Automatic staging for image processing. Number MIP-0410. 2004.