Type Analysis on Bitstring Expressions

Ralf Dreesen
University of Paderborn
Department of Computer Science
rdreesen@upb.de

Michael Thies
University of Paderborn
Department of Computer Science
mthies@upb.de

Uwe Kastens
University of Paderborn
Department of Computer Science
uwe@upb.de

Abstract

Bitstring expressions are well suited to describe dataflow of hardware and are, for instance, used in processor manuals to specify the semantics of instructions. To generate a hardware implementation from a bitstring expression, the width of functions and intermediate results must be known statically. As this information is not specified explicitly, it must be derived from the application context of functions.

In this paper, we present a type system and an inference algorithm that solves this task. To represent the width of a bitstring, two parameterized types are introduced. Another parameterized type is defined to couple literal arguments and type variables, which effectively enables generic integer parameters and thereby a way of explicit typing.

Type inference determines the values of type variables, by solving a system of equations and inequalities. Multiple solutions for type variables are allowed, as the expression's semantics are guaranteed to be solution invariant. We have successfully integrated the type system and the type inference algorithm in a generator that translates a processor specification into synthesizable VHDL code. We have evaluated our approach using a specification of the ARM architecture.

1. Introduction

Bitstring expressions are well suited to describe the data flow of hardware structures. This concept is for instance used in ISA (Instruction Set Architecture) manuals of processors [1, 8, 16] to define the semantics of instructions in a semi-formal manner. The width of bitstrings however is neither explicitly nor implicitly defined in these manuals.

The domain of bitstring expressions are sequences of bits, like `011010`. The expression `cat(10,010)` describes for instance the concatenation of the bitstrings `10` and `010`, yielding the bitstring `10010`. Besides `cat`, there are numerous other functions like addition (`add`) or sign extension (`exts`), which are defined on bitstrings.

All these functions operate on bitstrings and only on bitstrings. The `add` function is, for example, defined as the two's complement addition on bitstrings, not as the mathematical "+" operator on integers. The restriction of the domain to bitstrings yields a lean and well defined type system. Mixing several domains like bitstrings, integers and floating point values (as in ISP [2] and HML [11]) results in a complex language, which requires a large set of coercion rules.

Similar to [3, 5], we employ a type system to analyse a program property. The property is the width of bitstrings, which is needed to generate a hardware implementation. We have integrated the proposed type system into the ISA specification language ViDL (Versatile ISA Description Language). The proposed type inference algorithm is implemented in a generator, which translates a ViDL specification into synthesizable VHDL code. We have validated the type system and the type inference algorithm by generating VHDL code from a ViDL specification of the ARM architecture [1]. The results of this evaluation are discussed in Section 4. Besides VHDL code, the generator also produces an instruction set simulator (ISS) implemented in C.

1.1 Related Work

In Möller's work [14], bitstrings are used as the domain for systems of equations. The focus is on the solution of a system of equations, where the widths of bitstrings is one variable among others. The effect of bit width on semantics is essential for their approach. However, for a well defined specification of instruction semantics, this effect must not occur.

The parameterized types that we use in our typesystem to infere bit widths are related to the class of first order dependent types [12]. For general dependent types, type checking and type inference is undecideable. We therefore specialize our type system such, that index expressions are non-recursive and pure. This guarantees, that type inference is decideable and our algorithm can efficiently infer all bit widths at generation time.

There are many approaches to define the semantics of instructions. In the following, the languages and type systems of classical processor specification languages (PSLs), VHDL and HML will be regarded.

Processor Specification Languages In processor specification languages (PSL) such as nMl [4], ISDL [6], Lisa [7], UPSLA [10], and Expression [13], instruction semantics are specified by fragments of embedded C code. This C code can directly be copied into the simulation code of an instruction set simulator (ISS). There is no need for a type analysis by the ISS generator, as this task is solved by the C-Compiler.

The underlying type system is that of C, i.e. there is a fixed set of signed and unsigned integer types. Variables as well as functions are explicitly typed. Since the set of integer types is limited, instruction semantics must be broken down to these types and the respective operations. This includes the tedious splitting of long bitstrings and masking of bitstrings that do not match the width of C-integers. A bit-precise ISA specification language like ViDL eliminates this task, but requires a sophisticated type system and a non-trivial code generator.

Recent Lisa versions include a `Bit` datatype to model bitstrings precisely. Apparently, the type is not fully integrated into the Lisa tools, but simply mapped to a `C++` class. As a result, the bitwidth is determined either statically by the `C++` compiler or is even a dynamic property of a bitstring. There are some anomalies and limitations, that hint at this implementation of the `Bit` type:

- The Lisa manual states, that the use of the `Bit` type significantly reduces the simulation speed and should therefore be avoided in favor of C integer types.
- The manual also says, that the `WordWidth` operator (a kind of `widthof`) is missing for HDL generation.
- Muhammad et al. [15] report, that "The LISA language analyzer is quite limited.". Some specification errors are not detected by Lisa, but result in defective ISS code, which is ultimately rejected by the `C++` compiler.

In contrast, the ViDL generator performs full static type inference. All bitwidths are known by the generator and constant in the generated code. The above limitations do therefore not apply to ViDL and its generators.

VHDL Using the hardware description language VHDL [9], the semantics of a processor implementation are expressed with bit-precision. The bit widths of entities (functions) and signals (intermediate results) is defined by the developer. Splitting and masking as for C code is not necessary. VHDL code is typed explicitly, by specifying the widths of every (generic) entity and every signal. This lengthens the specification and may make changes of the datapath-width tedious.

HML Like ViDL, HML [11] is a functional language to describe the dataflow of hardware structures. In HML, the type analysis is performed directly on the abstract syntax tree of the HML specification. This requires a more complex type system to cover constructs like lambda expressions and variables. ViDL offers the same language constructs as HML (including polymorphic functions and higher order functions), but uses a much simpler type system. This is possible because the AST is first transformed, such that only bitstring expressions need to be regarded by the type inference. Another difference is, that HML uses higher level types like "integer", "bit", "bool" and "bit-vector", similar to ISP [2]. The relation of these types (in terms of coercion and subtyping) however is not described. In contrast to that, the philosophy of ViDL is to use a lean system of related (by means of subtyping) bitstring types.

1.2 ISA of Power Processors

Branch ***I-form***

b	target_addr	(AA=0 LK=0)
ba	target_addr	(AA=1 LK=0)
bl	target_addr	(AA=0 LK=1)
bla	target_addr	(AA=1 LK=1)

18	LI	AA	LK
0	6	30	31

```
if AA then NIA ←iea EXTS(LI || 0b00)
else       NIA ←iea CIA + EXTS(LI || 0b00)
if LK then LR ←iea CIA + 4
```

target_addr specifies the branch target address.

If AA=0 then the branch target address is the sum of LI || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value LI || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:

LR (if LK=1)

Figure 1. Definition of the branch instruction in the Power ISA manual [8].

As a real-world source for bitstring expressions, the Power ISA manual [8] is used in the following. Figure 1 shows an excerpt from the ISA manual, in which the semantics of the branch instruction are defined.

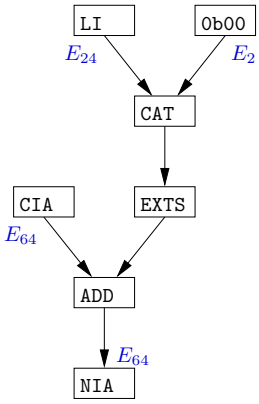

Figure 2. Kantorovitch tree of the target address computation.

The highlighted expression describes the computation of the branch target address for relative branches. In this expression, the bitstrings `LI` and `0b00` are concatenated (`||`) first. The result is then sign extended (`EXTS`) and finally added (`+`) to the contents of the register `CIA`, which contains the "Current Instruction Address". This sum is then written to register `NIA`, which represents the "Next Instruction Address". The domain of this specification is that of bitstrings, i.e. all operations operate on bitstrings only, not on integers. In the following, the functional notation

$$\texttt{NIA} = \texttt{ADD}(\texttt{CIA}, \texttt{EXTS}(\texttt{CAT}(\texttt{LI}, 0b00)))$$

is used, to denote the computation of the bitstring expression. For illustration purposes, a Kantorovich tree as shown in Figure 2, is used.

1.3 Hardware Implementation

The Kantorovitch tree of the bitstring expression can be used as the dataflow graph (DFG) of a hardware implementation. To implement the DFG in Figure 2 in hardware, the width of functions and intermediate results must be known. As hardware is inherently static, these widths must be determined statically. For the `EXTS` function in Figure 2, for example, the width of the argument and of the result must be determined. The width of the argument is 26 bits, which is the sum of the widths of the `LI` field (24 bits) and the literal `0b00` (2 bits). As the registers `NIA` and `CIA` are 64 bits wide, the `add` function must operate on 64 bits and hence, the result of `EXTS` must be 64 bits wide. To derive the widths of functions and intermediate results systematically, we use type inference.

2. Type System

The type system, that is presented in this section is the result of an extensive exploration on how ISAs can be specified clearly, efficiently and unambiguously. Besides, the type system should be simple and understandable on the one hand and powerful enough to specify real-world ISAs on the other hand.

In the following sections, we first define a set of parameterized types (2.1), which model bit width and are in subtype relation (2.2). Using these types, bit width constraints of functions can be formulated by polymorphic signatures (2.3). This includes functions that implicitly truncate their actual arguments (2.4) to support resource sharing. Using a parameterized type for integer literals, even signatures can be expressed, that effectively bind arguments to type variables (2.5). To demonstrate the power of the type system, some representative and non-trivial signatures are finally discussed (2.6).

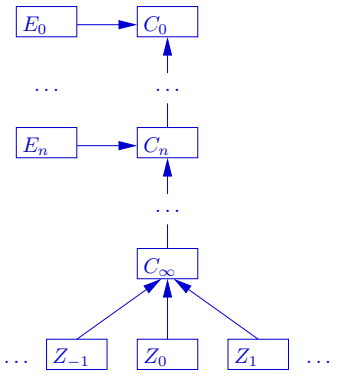

Figure 3. Type hierarchy.

2.1 Bitstring Types

The width of a bitstring is represented by a type. Figure 3 shows the hierarchy of types used. The types E_x, C_x and Z_x are parameterized types . For E_x and C_x, the type variable $x \in \mathbb{N}_0$ is a non-negative integer. For Z_x, the type variable $x \in \mathbb{Z}$ is an integer.

E_x type: The parameterized type E_x represents the set of bitstrings that are exactly x bits wide.

$$E_x = \{0,1\}^x$$

The set of values of E_2 is for example $E_2 = \{00, 01, 10, 11\}$. Two types $E_x, E_y, x \neq y$ are disjoint and thereby not in subtype relation. The type E_0 contains only the bitstring of length 0, which is the empty word ϵ. As the type contains only a single element, a value of that type carries no information.

C_∞ type: The type C_∞ represents the set of bitstrings with an infinite width. In mathematics, this set is known as Cantor set (denoted $2^{\mathbb{N}}$ or 2^{ω}) which is homeomorphic to the set of functions $\mathbb{N}_0 \mapsto \{0,1\}$. In the following we represent an infinite bitstring of C_∞ by such a function, which maps each bit position to a bit value.

C_x type: The parameterized type C_x represents the set of bitstrings that are *at least* x bits wide. The set contains bitstring with a finite width $i \geq x$ and bitstrings of infinite width. The C type is defined using the E type and the C_∞ type.

$$C_x = \bigcup_{i \in \mathbb{N}_0, i \geq x} E_i \cup C_\infty$$

The set of C_2 contains for example the bitstrings $C_2 = \{00, 01, 10, 11, 000, 001, \ldots\}$. Note, that the set is infinite.

Z_x type: The third parameterized type Z_x is the type of the integer literal $x \in \mathbb{Z}$ and only of this integer literal. Z_x is a singleton set containing the bitstring representation of the integer x, according to its two's complement representation.

$$Z_x = \{\text{twosComp}_x : \mathbb{N}_0 \mapsto \{0,1\}\}$$

The bitstring of an integer has an infinite width. The bitstring is therefore defined by a function $\text{twosComp}_x(i)$, which yields the i-th bit of the bitstring. The function twosComp_x is defined with respect to the two's complement.

$$\text{twosComp}_x(i) = \begin{cases} \left\lfloor \frac{x}{2^i} \right\rfloor \bmod 2 & x \geq 0 \\ 1 - \left\lfloor -\frac{v+1}{2^i} \right\rfloor \bmod 2 & x < 0 \end{cases}$$

In addition to these bitstring types, types for tuples of bitstrings are defined.Tuple types allow for a more formal discussion of the type system.

2.2 Subtyping

The parameterized types E_x, C_x, Z_x and the type C_∞ are in subtype relation. According to the definition of the C type, C_{x+1} and E_x are subtypes of C_x. In other words, the set of bitstrings of C_x is a superset of E_x and C_{x+1}. As the subtype relation is reflexive and transitive, any type E_y resp. C_y is a subtype of C_x when $x \leq y$. The subtype relation expresses that, wherever a bitstring of *at least* x bits is expected, a bitstring of exact or minimal length $y, y \geq x$ may be given. If a bitstring of *exactly* x bits is expected, a bitstring of exactly x bits must be supplied.

As the type C_∞ contains the bitstrings of infinite width, it is a subtype of any $C_x, x \in \mathbb{N}_0$. In other words: wherever a bitstring of minimal width x is expected, a bitstring of infinite width may be given.

The type Z_x is a subtype of C_∞. The type Z_x contains only the infinitely wide bitstring of the integer literal x, which is also a value of C_∞. In the domain of bitstrings this means that wherever a bitstring of minimal width y is expected, an integer literal may be given instead. In addition, the integer literal x may be given, where the type Z_x is expected. This property of the type system is used in Section 2.5 to couple arguments and type parameters.

The type C_0 is the top element of the type hierarchy. That type represents all bitstrings of at least 0 bits, which is the set of all bitstrings $\{0,1\}^* \cup \{0,1\}^\infty$. The type hierarchy does not include a bottom element. The set of values of the bottom type would be empty, as two different E_x types are disjoint.

2.3 Signatures

In the previous section, the type hierarchy was introduced, including three parameterized types . These types are now used to define the signature of functions. Most functions are parametrically polymorphic, as they are defined for arbitrary bit widths. Each function introduces a set of type variables, which are used in parameterized types of the signature. The concatenation function `CAT` for instance concatenates a bitstring of exactly x bits and a bitstring of exactly y bits to a bitstring of exactly $x + y$ bits. This is reflected by the signature $\texttt{CAT}_{x,y} : E_x \times E_y \mapsto E_{x+y}$, where x and y are type variables of `CAT`.

As we use implicit typing, the type variables x and y are inferred from the application context of `CAT`. In the example in Figure 2, the type of `LI` is known to be E_{24} and the type of `0b00` to be E_2. According to the signature, the result type of `CAT` must be E_{26}. This is just a small example to give an idea of the type inference, which is covered in depth in Section 3.

An important property of implicit typing is that type parameters are inferred and need not be defined by the developer. This is different from explicit typing, where the developer has to specify the values of type variables along with the instantiation. Explicit typing increases the complexity of the specification and reduces the maintainability. This effect is discussed in the evaluation in Section 4. Implicit typing also allows for optimization of the datapath, but requires an additional analysis phase in the generator.

2.4 Implicit Truncation

To simplify the specification and support the optimization of the hardware implementation, some functions perform an implicit truncation on arguments. A truncation $\texttt{TR}_x(a)$ yields the x least significant bits of the argument a and discards any excessive most significant bits. Figure 4 shows an example of implicit truncation. The $\texttt{ADD}_x$ function truncates the arguments to x bits before the addition is performed. The semantics of the truncating $\texttt{ADD}_x$ function are defined as

$$\texttt{ADD}_x(A, B) := \texttt{ADD}_x^{NT}(\texttt{TR}_x(A), \texttt{TR}_x(B))$$

where $\texttt{TR}_x : C_x \mapsto E_x$ truncates a bitstring of at least x bits to exactly x bits and $\texttt{ADD}_x^{NT} : E_x \times E_x \mapsto E_x$ represents a non-truncating x bit adder. From the signature of `TR` and $\texttt{ADD}^{NT}$, the signature of the truncating addition can be derived as $\texttt{ADD}_x : C_x \times C_x \mapsto E_x$. To ensure unambiguous semantics of any bitstring expression, the result type

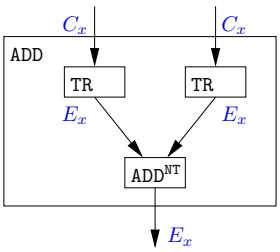

Figure 4. Implicit truncation of ADD's parameters.

is weakened to be C_x, accepting a loss of precision in typing. The rules for unambiguous semantics are discussed in Section 3.1 in detail. The final signature of ADD is

$$\texttt{ADD} : C_x \times C_x \mapsto C_x$$

This signature expresses, that the ADD function expects two bitstrings of at least x bits and yields a bitstring of at least x bits.

Implicit truncation is only allowed for functions with certain semantics. The parameter of the EXTS function must for example not be truncated implicitly. This would lead to ambiguous semantics, if the typing is ambiguous. This phenomenon is discussed in detail in Section 3.1.

2.5 Coupling of Actual Parameters and Type Variables

There are some functions, for which the signature depends on an argument. An example is the ONES(A) function, which returns a bitstring of exactly A '1'-bits. The result of ONES(5) is for example the bitstring 11111 of type E_5. The type parameter x of the result type E_x is effectively given by the argument A.

As the hardware implementation relies on static typing, the argument of ONES must be a static value. If the parameter would be dynamic, the result type and thereby the width of the signal would be dynamic.

Exploiting, that a constant argument x has the type Z_x, the signature of ONES is

$$\texttt{ONES} : Z_x \mapsto E_x$$

The type parameter type Z_x restricts arguments to integer constants. A dynamic argument will be reported as a type conflict.

Basically, the parameterized type Z_x couples constant arguments with type variables. This way, a value for a type parameter x can be specified explicitly by a (formal) parameter of type Z_x. A nice example to demonstrate the power of this paradigm is the CUT function. The CUT(V, X, Y) function extracts a bit slice from bit X to bit Y from the bitstring V. The application CUT$(111000, 4, 2)$ yields for example the bitstring 110 of type E_3. The signature of CUT is defined as

$$\texttt{CUT} : C_{X+1} \times Z_X \times Z_Y \mapsto E_{X-Y+1}$$

The type variables X and Y are conceptually defined by the second and third argument in the application of CUT. These two parameters therefore resemble generic integer parameters. If the type system would not include the parameterized type Z, a concept like generic parameters would be necessary to model the signature of CUT. However, the combination of parametric polymorphism and the parameterized type Z supersedes the need of generic parameters.

2.6 Signatures of Typical Functions

In addition to signatures already mentioned, further signatures of representative functions are listed in Figure 5 and discussed in the following. The EXTZ (zero-extension) and EXTS (sign-extension) functions for instance extend a bitstring of exactly x bits to a bitstring of at least y bits, where y is independent of x. The result can therefore have an arbitrary width. In the DFG in Figure 2, the EXTS function extends a bitstring of exactly 26 bits to a bitstring of at least 64 bits.

$$\begin{aligned}
\texttt{ADD}_x &: C_x \times C_x \mapsto C_x \\
\texttt{CAT}_{x,y} &: E_x \times E_y \mapsto E_{x+y} \\
\texttt{EXTZ}_{x,y} &: E_x \mapsto C_y \\
\texttt{EXTS}_{x,y} &: E_x \mapsto C_y \\
\texttt{ROTR}_{x,y} &: E_x \times E_y \mapsto E_x \\
\texttt{CUT}_{x,y} &: C_{x+1} \times Z_x \times Z_y \mapsto E_{x-y+1} \\
\texttt{TR}_{x,y} &: C_x \times Z_x \mapsto E_x \\
0ba_{x-1} \ldots a_0 &: \mapsto E_x \\
\texttt{Reg} &: \mapsto E_{\text{width(Reg)}} \\
\texttt{RegAssign} &: C_{\text{width(Reg)}} \mapsto \text{Unit} \\
\texttt{Imm} &: \mapsto E_{\text{width(Imm)}} \\
< \texttt{IntLit} > &: \mapsto Z_{<\texttt{IntLit}>}
\end{aligned}$$

Figure 5. Signatures of polymorphic functions.

A bitstring literal is regarded as a constant function, where the result type expresses the width of the literal. The constant function of the literal $0b00$ has for example the signature $0b00 :\mapsto E_2$. The type C_2 (which is a supertype of E_2) would also be a valid result type, but is less precise and can therefore not be used, where the type E_2 is expected.

A "using" occurrence of a register in a bitstring expression is also modeled by a constant function. The width of the result is given by the width of the register. The function is therefore not polymorphic. For the CIA register, the signature of the function is CIA $:\mapsto E_{64}$.

An assignment to a register is modeled by a unary function which expects a bitstring that has at least the width of the register. Wider bitstrings are implicitly truncated to the width of the register. An assignment has no result, which is expressed by the result type "Unit $= \{()\}$", as known from SML and other functional languages. For the NIA register for instance, the assignment function has the signature NIA $: C_{64} \mapsto$ Unit.

An immediate instruction operand is modeled by a constant function, whose result type corresponds to the width of the immediate. For the LI immediate in the example, the signature is LI $:\mapsto E_{24}$.

An integer literal x is also modeled by a constant function, where the result type Z_x represents the value of the integer. The signature of the integer literal 4 is for example $4 :\mapsto Z_4$.

3. Type Inference

The type inference algorithm determines feasible values for the type variables of function applications. The values are selected such that the bitstring expression is properly typed. That is the case, if the type of each argument is a subtype of the respective parameter type.

Hence, the input for the type inference is a set of constraints, where each constraint is a subtype relation between two types.

For the inference, the set of subtype constraints is transformed into a system of equations, which is then solved by the generator. Table 1 shows the translations of subtype constraints into equations, as derived from the type hierarchy in 3. The constraint $E_x <: C_y$ is, for example, transformed into the equivalent inequality $x \geq y$. The constraint $Z_x <: C_y$ always holds and is therefore transformed into the statement "True", which means, that the constraint does not contribute an equation. The constraint $C_x <: E_y$ never holds and is therefore transformed into the statement "False", which means, that the type constraints are contra-

$T_1 <: T_2$		T_2 C_y	E_y	Z_y	C_∞
T_1	C_x	$x \geq y$	False	False	False
	E_x	$x \geq y$	$x = y$	False	False
	Z_x	True	False	$x = y$	True
	C_∞	True	False	False	True

Table 1. Equivalent equations for subtype relations.

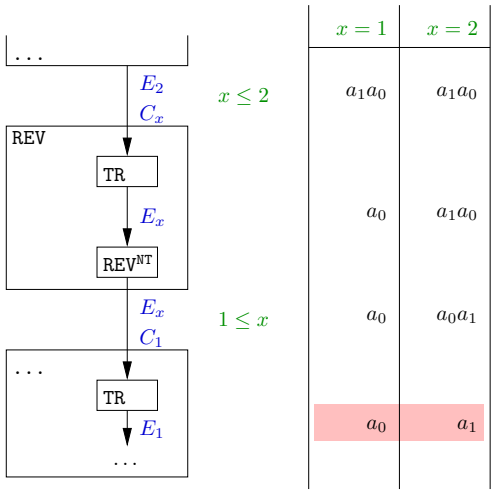

Figure 6. An ambiguous typing of the `REV` function implies ambiguous semantics.

dictory. There is no valid typing of the bitstring expression in this case.

If there is no contradictory subtype constraint, the transformation yields a system of equations, which is equivalent to the set of subtype constraints. Further inequalities are added to the system of equations, to ensure that the type parameters of E and C types are non-negative. For an application of $\texttt{CUT} : C_{X+1} \times Z_X \times Z_Y \mapsto E_{X-Y+1}$ for instance, the inequalities $X + 1 \geq 0$ and $X - Y + 1 \geq 0$ are added.

This system of equations is then solved by an equation solver. If it has no solution, the bitstring expression can not be typed properly, which means that the bitstring expression is not correct. If it has exactly one solution, the typing is unique. If the system of equations has multiple solutions, the type inference algorithm selects one solution. Our theory ensures that the bitstring expressions of all solutions are semantically equivalent (Section 3.1). The inference exploits this degree of freedom, to optimize the datapath of the bitstring. In particular, the sharing of resources in the hardware implementation is maximized, by merging common subexpressions in the DFG. An aggressive optimization of the datapath-width may be in conflict with this optimization goal.The next section shows that the semantics of a bitstring expression are equivalent for all solutions.

3.1 Invariant Semantics

The type system may have multiple solutions, in which case the type inference algorithm selects a solution based on an optimization criterion. Which solution is chosen is not defined in the language and is therefore undefined from the developers perspective. The semantics of the bitstring must therefore be the same for all solutions.

The example in Figure 6 demonstrates ambiguous semantics for an application of the `REV` function. The $\texttt{REV} : C_x \mapsto E_x$ function reverses the bits of a bitstring. In the example, an argument of type E_2 is given and a result of type C_1 is expected. To be properly typed, the inequality $1 \leq x \leq 2$ must hold. For the solution $x = 1$, the least significant bit (LSB) of the result is the LSB a_0 of the argument, but for $x = 1$ it is bit a_1 of the argument. As the selection of the solution for x is undefined, the result of the expression is undefined. The origin of this ambiguity lies in the semantics of `REV` and the implicit truncation of its argument. If the signature is chosen to be $\texttt{REV} : E_x \mapsto E_x$, there is only one solution for x $(1 \leq x = 2)$ and the semantics are thereby unique.

In contrast to `REV`, implicit truncation of the $\texttt{NOT} : C_x \mapsto C_x$ function does not lead to ambiguous semantics because `NOT` is a bitwise function. In general, solution invariant semantics are guaranteed by two conditions.

- A specific dependency between the type variables of parameter types and the type variables of result types (*E type derivation*).
- A specific relation between the signature and the semantics of a function (*invariant function semantics*).

The resulting invariance of semantics is proven later in this paper.

E Type Derivation This condition demands that the type parameter of an E_x result is uniquely defined by the E types of the parameters. This informal description of the condition can formally be expressed as follows: For the function f

$$f_{x_1,\ldots,x_k} : E_{g_1(x_1,\ldots,x_k)} \times \ldots \times E_{g_n(x_1,\ldots,x_k)} \times C_{\ldots} \times \ldots \times C_{\ldots} \mapsto E_{h(x_1,\ldots,x_k)}$$

and constants a_i, the system of equations

$$\begin{aligned} g_1(x_1,\ldots,x_k) &= a_1 \\ &\vdots \\ g_n(x_1,\ldots,x_k) &= a_n \\ h(x_1,\ldots,x_k) &= b \end{aligned}$$

must have at most one solution for b.

Invariant Function Semantics The second condition demands that, for a given context of a function application, all valid instances of the function must have the same semantics.

$$\begin{aligned} &\forall \hat{A}, \hat{B} : \\ &\forall \hat{a} \in \hat{A} : \\ &\exists \hat{b} \in \hat{B} : \\ &\forall f : A \mapsto B <: \hat{A} \mapsto \hat{B} : f(\hat{a}) \simeq_{\hat{B}} \hat{b} \end{aligned}$$

The context of the function is given by the type of the argument $\hat{A}$ and the expected result type $\hat{B}$. For this context, a specific argument $\hat{a}$ must be mapped to the same value $\hat{b}$ by all instances $f : A \mapsto B$, which are subtypes of the expected signature $\hat{A} \mapsto \hat{B}$. The signature is a subtype of the expected signature, if the parameter type is contravariant $(\hat{A} <: A)$ and the result type is covariant $(B <: \hat{B})$.

To be precise, the function result and the value $\hat{b}$ need not necessarily be equal. The result and $\hat{b}$ need only be in $\simeq_{\hat{B}}$ relation, which is weaker than the equal relation.

$$u \simeq_{\hat{B}} v := \begin{cases} \texttt{TR}_x(u) = \texttt{TR}_x(v) & \text{for } \hat{B} = C_x \\ u = v & \text{otherwise} \end{cases}$$

If $\hat{B}$ is a type C_x, only the x least significant bits must be equal. That is because, the result is truncated by any succeeding function to at most x bits. Any other bits are discarded and need therefore not be equal.

These constraints only regard the signature and semantics of functions. The developer of a primitive must define the signature such, that the constraints hold. The constraints need not be considered by the user of primitives and need not be checked during type inference.

3.2 Proof of Solution Invariance

In the following, we briefly outline the proof of solution invariance for all intermediate values. An intermediate value a_i resp b_i is solution invariant, if it is a substring of a fixed value a^∞ for all solutions $s \in \text{TSol}$ of the type system.

$$\exists a_i^\infty : \forall s \in \text{TSol} : a_i^s \simeq_{A_i^s} a_i^\infty$$
$$\exists b_i^\infty : \forall s \in \text{TSol} : b_i^s \simeq_{B_i^s} a_i^\infty$$

Solution invariance of E/Z-Types In the following, only E types are regarded, but the same applies to Z types. We first show, that the type parameter of each E-type is solution invariant. This is proven by induction over the topological order of function applications f_i. The induction basis holds, as f_1 is an application of a constant function. The type-parameter x of each E-type result must therefore be constant according to the *E type derivation* condition. For an application f_i, each E parameter is invariant, as the corresponding E result of f_j, $j < i$ is invariant. Since all E parameters are invariant, each E result of f_i is invariant accoring to condition 1.

Solution Invariant Subtype T^∞ To show the solution invariance of values, we first define the smallest subtype T^∞ of a given type T. The type T^∞ is a common subtype of all T^s for $s \in \text{TSol}$, as E/Z-Types are invariant and C^∞ is a subtype of any type C_x.

$$T = T_1 \times \ldots \times T_n$$
$$T^\infty = T_1^\infty \times \ldots \times T_n^\infty$$
$$T_i^\infty = \begin{cases} C^\infty & \text{for } T_i = C_x \\ T_i & \text{otherwise} \end{cases}$$

Solution Invariant of Intermediate Values The solution invariance of the values a_i, b_i is shown by induction over the topological order of f_i. The induction basis holds for a_1, as f_1 is constant and A_1 is Unit. As a_i is composed of results $b_j, j < i$ which are invariant, a_i is invariant. The next section shows, that b_i is invariant, if a_i is invariant.

Solution Invariant Result As we only regard a single function application f_i in the following, the index i of a_i and b_i is omitted. We proof by contradiction that b is invariant, by assuming that b is not invariant. This means, that b^∞ does not exists. Hence, there must be two solutions $r, s \in \text{TSol}$, such that

$$b^\infty \simeq_{\hat{B}^r} b^r \wedge b^\infty \simeq_{\hat{B}^s} b^s$$

does not hold for any b^∞. Let $\hat{B}$ be the smallest common supertype of B^r and B^s. Then, the values b^r and b^s are not equal with respect to $\hat{B}$.

$$b^r \not\simeq_{\hat{B}} b^s$$

From the *Invariant Function Semantics* condition we can infer, that for a result type $\hat{B}$ all valid function instances f yield an invariant result.

$$\exists \hat{b} \in \hat{B} :$$
$$\forall f : A \mapsto B <: A^\infty \mapsto \hat{B} : f(a^\infty) \simeq_{\hat{B}} \hat{b}$$

As f^r and f^s are functions of type $A^\infty \mapsto \hat{B}$, the results b^r and b^s must be equal with respect to $\hat{B}$

$$\exists \hat{b} \in \hat{B} : f^r(a^\infty) \simeq_{\hat{B}} \hat{b} \wedge f^s(a^\infty) \simeq_{\hat{B}} \hat{b}$$
$$\Rightarrow b^r \simeq_{\hat{B}} b^s$$

This is a contradiction to $b^s \not\simeq_{\hat{B}} b^t$, which was inferred from the assumption that b is not invariant. Therefore b must be invariant. Hence, all values in the DFG are invariant and the semantics of the DFG do therefore not depend on the solution that is selected by the type inference.

```
NIA= if M32 then EXTZ(TR(
          if AA then EXTS(CAT(LR,0b00))
                else ADD(CIA,EXTS(CAT(LR,0b00)))
                                              ,32))
     else if AA then EXTS(CAT(LR,0b00))
                else ADD(CIA,EXTS(CAT(LR,0b00)))
if LK then LR = ADD(CIA,4)
```

Figure 7. Specification of semantics of Power ISA Branch.

4. Evaluation

In the following, we evaluate the type system of ViDL using the ARM and Power ISA. First, we regard the branch of the Power ISA to demonstrate, how to specify non-trivial, real world instructions in ViDL. The specification is simple, compact and intuitive. We then show that flaws in the specification such as ambiguous semantics are detected by our type inference algorithm. For instance, we found a non-obvious ambiguity in the ARM ISA manual. We demonstrate, how implicit truncation increases hardware sharing and reduces the width of the datapath using the example of the ARM `add` instruction. Finally, we analyze the results of the type inference to show, that implicit typing and implicit truncation eliminate amounts of type annotations and many applications of explicit truncation.

Branch of Power ISA in ViDL Figure 7 shows the specification of a real world example in the ISA specification language ViDL. The two bitstring expressions specify the behavior of the Power branch instruction shown in Figure 1, including relative and absolute branches in 32 and 64 bit mode, as well as the optional setting of the link register.

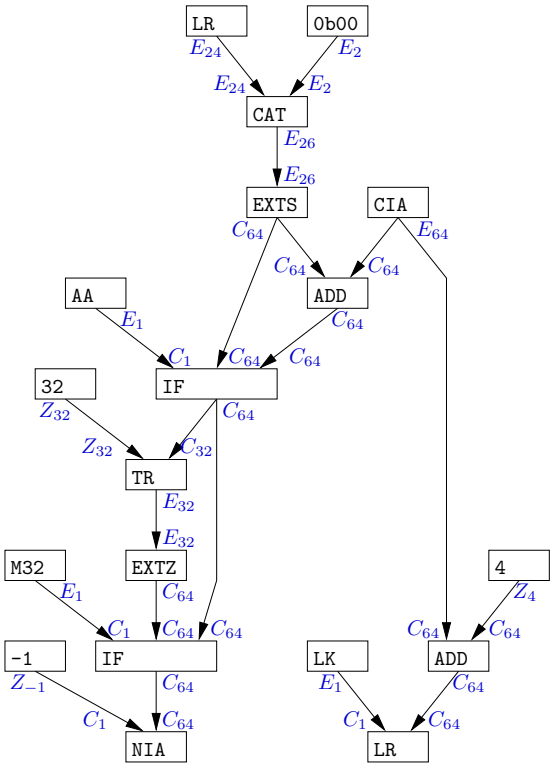

Figure 8. Optimized and typed DFG of Power ISA branch.

Figure 8 shows the resulting DFG of the example, annotated with all inferred types. Signatures are selected such, that common subexpressions in the code in Figure 7 can be merged in the DFG. The example demonstrates, how easily a complex real-world instruction can be described in ViDL. The complexity of width assignment, implicit truncation and hardware sharing is moved from the user to the type inference of the generator. After further optimizations by the

Operation

```
if ConditionPassed(cond) then
    if opcode[25] == 1
        operand = 8_bit_immediate Rotate_Right (rotate_imm * 2)
    else /* opcode[25] == 0 */
        operand = Rm
    if R == 0 then
        if field_mask[0] == 1 and InAPrivilegedMode() then
            CPSR[7:0] = operand[7:0]
        if field_mask[1] == 1 and InAPrivilegedMode() then
            CPSR[15:8] = operand[15:8]
        if field_mask[2] == 1 and InAPrivilegedMode() then
            CPSR[23:16] = operand[23:16]
        if field_mask[3] == 1 then
            CPSR[31:24] = operand[31:24]
    else /* R == 1 */
```

Figure 9. Excerpt from ARM ISA manual [1], page A4-62.

generator, the typed DFG is used to generate VHDL code and C code for an ISS.

Ambiguous semantics in ARM Manual Figure 9 shows the semantics of the MSR instruction, as specified in the ARM ISA manual. The highlighted expressions describes the rotation of an 8 bit immediate, followed by an extraction of bits 31 to 24. The result of the rotation must therefore at least be 32 bit wide.

Although the semantics of the expression seem to be sound, the application of the rotation is not well defined. The width of the argument suggests an 8 bit wide rotation, whereas the expected result suggests a 32 bit wide rotation. In both cases, either the argument or the result must be extended. Which kind of extension should be applied (sign or zero extension) is not specified. Two aspects of the instruction are therefore undefined: The width of the rotation and the kind of extension.

In ViDL, the combination of both expressions is denoted

$$\texttt{CUT}(\texttt{ROTR}(\texttt{imm8}, \texttt{CAT}(\texttt{imm4}, 0b0)), 31, 24)$$

The ambiguity of this expression is detected by type inference. According to the signatures in Figure 5, the result type of `ROTR` is E_8, but the expected type for the first parameter of `CUT` is C_{32}. As E_8 is not a subtype of C_{32}, a type error is reported. This notifies the designer to correct the expression by inserting a proper sign extension. The following expression precisely defines the semantics, as intended by the ARM manual:

$$\texttt{CUT}(\texttt{ROTR}(\texttt{TR}(\texttt{EXTZ}(\texttt{imm8}), 32), \texttt{CAT}(\texttt{imm4}, 0b0)), 31, 24)$$

Type analysis on the ARM ISA The ViDL specification of the ARM instruction set consists of 750 lines, compared to 260 pages in the instruction manual. The ViDL specification contains about 40 explicit truncations to uniquely define ISA semantics. The frontend of the generator transforms the ViDL specification into a DFG, on which the following results are based.

The DFG consists of about 1500 function applications, which introduce about 1600 type variables. About 20% of all function applications contribute no type variable, 54% contribute one type variable and 26% contribute two variables. On average, each function application contributes one type variable. The type inference phase of the generator constructs and solves the equation system in less than two seconds.

The 1600 derived type variables define about 4550 types (approx. 3 types per function) in the DFG, of which 60% are C types, 29% are E types and 11% are Z types. The percentage of C types in the DFG is remarkably high, which facilitates the application of implicit truncation in the DFG.

There are 2200 positions in the DFG, where implicit truncation may legally be applied. A real implicit truncation, where the bitstring is actually narrowed, is applied at 140 positions (6.6%) in the DFG. Figure 10 shows the number of implicit truncations for the width before and after truncation.

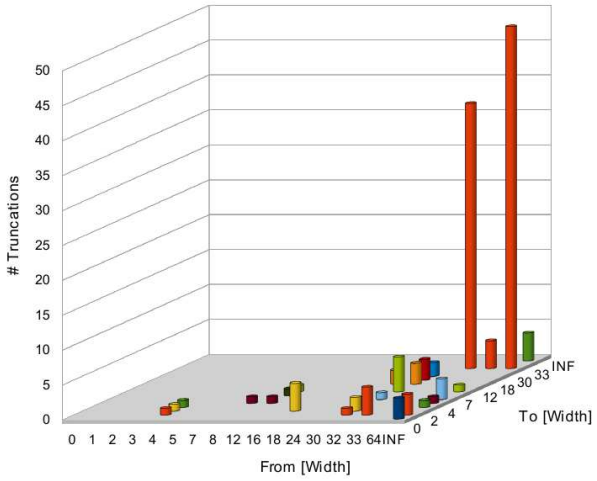

Figure 10. Number of implicit truncations from a given bit width to a given bit width.

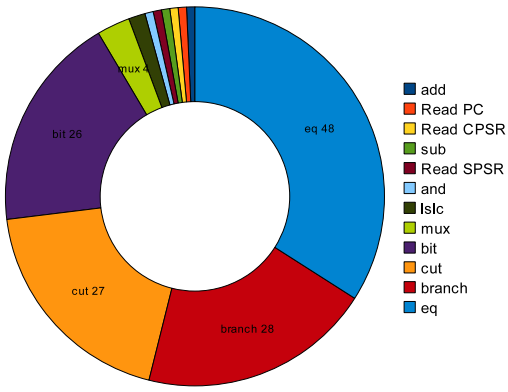

Figure 11. Distribution of implicit truncations among functions.

The frequent truncation from infinite bitstrings (INF) to 32 bit is a result of using integers, where a bitstring of at least 32 bits is expected (e.g. `EQ(register,0)`).

The implicit truncations from 33 bits to 32 bits are a result of resource sharing. An `ADD` instruction for example includes the following two definitions.

```
carry_flag  := bit(add(a,b),32)
gp_register :=     add(a,b)
```

Due to resource sharing, only one adder is instantiated, which adds `a`, `b` and yields a 33 bit result. The most significant bit of the result is assigned to the carry flag. For the assignment to the GP register, the 33 bit result is implicitly truncated to 32 bits. Further 33 bit computations, which result in implicit truncation include subtraction, left-shift, rotation and the addressing modes of the second operand.

Figure 11 shows the distribution of implicit truncations among functions. The 4 most prevalent contributors are the `eq`, `branch`, `cut` and `bit` functions. For the `EQ` : $E_x \times C_x \mapsto E_1$ function, the second parameter is implicitly truncated to the width of the first parameter. The `branch` function is a two-way multiplexer, which implicitly truncates parameters of different widths. For the `BIT` and `CUT` functions, the occurrence of real implicit truncation expresses, that the argument is wider than required for the extraction.

Exploration of ISA Width To demonstrate, that implicit typing increases the maintainability of the specification, we generalized the ARM ISA from 32 bit to $n \geq 32$ bits. For instance, we generated 64, 65 and 256 bit wide implementations of the ARM processor. To change the width

of the ISA, we only changed the width of the general purpose register file. The width of the datapath (including the shifter-operand) is adjusted automatically, as it is inferred by type inference.

To provoke conflicts, we did not extend the width of the program counter, status register and memory. Instructions, which transfer data between these 32-bit storages and the n-bit general purpose registers therefore need to be considered. Fortunately, these transfers are identified and reported by the type inference as a type error. For the ARM ISA, this is the "branch and link" instruction, the MSR[1] instruction and the load instruction. We decided to apply a zero extension, which defines the most significant bits of the wider general purpose register to be zero. After the insertion of these extensions (which took about 15 minutes) the width of the ARM ISA can be widened as desired in an instant, by redefining the width of the general purpose registers. For the special case $n = 32$, the extensions have no effect, i.e. our specification defines a standard 32-bit ARM.

Thanks to implicit truncation, transfers from an n-bit general purpose register to a 32 bit register need not be considered. The n-bit value is automatically truncated to 32 bits.

5. Beyond Type Analysis

In this section, we briefly present some performance statistics, to show that our generator does not only correctly infer types, but also produces efficient C and VHDL code. Actually, this is beyond the scope of the paper, but we want to show that the type system is part of a practical generator.

We have evaluated the generated ARM instruction set simulator (ISS) on a standard 3 GHz Intel Pentium workstation. The average simulation speed is 70 MIPS, i.e. 70 million ARM instructions are decoded and simulated per second. The generated ISS of the 256-bit wide ARM ISA still runs at an average speed of 20 MIPS, although 256-bit arithmetic is broken down to 64-bit host arithmetic by the generator.

The optimization of the VHDL code generator is subject to current research, but first experiments are promising. We have evaluated the quality of the generated VHDL code for the ARM ISA on a 65 nm STMicroelectronics low power technology. Under worst case conditions we yield a clock frequency of 298 MHz, which is limited by a 64 bit multiplier. After removing the wide muliplication instructions from the ARM ISA and deactivating the bypass, we even yield a clock frequency of 526 MHz under worst case conditions. For a low power technology under worst case conditions, that clock frequency is in an acceptable range.

Both the ISS and the microarchitectural implementation are generated from the same ISA specification. The entire C and VHDL code is generated, i.e. no line of C or VHDL code has been added or tuned in any way.

6. Conclusion

We have proposed a type system and a matching type inference algorithm for bitstring expressions, which have been successfully integrated into the specification language ViDL and a respective VHDL and C generator. The type system precisely models the width of bitstring expressions as opposed to C-based PSLs. Implicit typing by inference shifts the tasks of explicit width-assignment from the developer to the generator.

Implicit truncation simplifies the specification and allows for resource sharing, by selecting an appropriate solution from the set of valid typings. An ambiguous typing is feasible, as the semantics of all solutions are guaranteed to be the same. The bit width constraints of an extensive set of 35 primitive functions have been defined by parametrically polymorphic signatures. This includes functions, where type parameters are in effect given by constant arguments by utilizing the Z_x type.

Our approach has been evaluated by specifying the ARM instruction set in ViDL, without the need for any explicit type annotations. Implicit typing not only simplifies the specification, but also increases the maintainability, as demonstrated by widening the ARM ISA. Results of the type inference show that the typing of our specification is ambiguous and implicit truncation is applied in 6.6% of 2200 cases. During our evaluation, the type inference identified a non-obvious ambiguous specification in the ARM manual by reporting a type error.

Future work concentrates on the development of ViDL and its generators. The proposed type system is just one important component of this system.

[1] MSR copies the content of the Status registers CPSR and SPSR to a general purpose register

References

[1] ARM Limited. *ARM Architecture Reference Manual*, ARM DDI 0100E edition, 2000.

[2] C. Gordon Bell and Allen Newell. The PMS and ISP descriptive systems for computer structures. In *AFIPS '70 (Spring): Proceedings of the May 5-7, 1970, spring joint computer conference*, pages 351–374, New York, NY, USA, 1970. ACM.

[3] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 56–69, New York, NY, USA, 2001. ACM.

[4] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. pages 503 –507, mar. 1995.

[5] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, 2003.

[6] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. pages 299 –302, jun. 1997.

[7] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1338 –1354, nov. 2001.

[8] IBM. *Power ISA*, Version 2.04 edition, April 2007.

[9] IEEE. *IEEE 1076-1993: Standard VHDL Language Reference Manual*. Institute of Electrical and Electronics Engineers, 1993.

[10] Uwe Kastens, Dinh Khoi Le, Adrian Slowik, and Michael Thies. Feedback driven instruction-set extension. In *Proceedings of ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, Washington, D.C., USA, June 2004.

[11] Yanbing Li and Miriam Leeser. HML, a novel hardware description language and its translation to VHDL. *IEEE Trans. Very Large Scale Integr. Syst.*, 8(1):1–8, 2000.

[12] James McKinna. Why dependent types matter. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 1–1, New York, NY, USA, 2006. ACM.

[13] Prabhat Mishra, Arun Kejariwal, and Nikil Dutt. Rapid exploration of pipelined processors through automatic generation of synthesizable RTL models. *Rapid System Prototyping, IEEE International Workshop on*, 0:226, 2003.

[14] M. Oliver Möller. Solving bit-vector equations - a decision procedure for hardware verification, 1998. Diploma Thesis, available at `http://www.informatik.uni-ulm.de/ki/Bitvector/`.

[15] Rashid Muhammad, Ludovic Apvrille, and Renaud Pacalet. Evaluation of ASIPs design with LISATek. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5114 of *Lecture Notes in Computer Science*, pages 177–186. Springer Berlin / Heidelberg, 2008.

[16] SPARC International, Inc. *The SPARC Architecture Manual*, SAV080SI9106 version 8 edition, 1992.