

Short Communication

TYPE-SAFE CASTING

WILSON C. HSIEH

*Dept. of Computer Science, Univ. of Utah, 50 S Central
Campus Drive, Room 3190, SLC, UT 84112*

MARC E. FIUCZYNSKI

*Dept. of Computer Science and Engineering, Univ. of
Washington, Box 352350, Seattle, WA 98102*

PRZEMYSŁAW PARDYAK

*Dept. of Computer Science and Engineering, Univ. of
Washington, Box 352350, Seattle, WA 98102*

BRIAN N. BERSHAD

*Dept. of Computer Science and Engineering, Univ. of
Washington, Box 352350, Seattle, WA 98102*

SUMMARY

Many modern extensible systems, such as Java and the *SPIN* operating system, depend on type safety for memory protection. Unfortunately, current type-safe languages do not support systems programming well, because they do not give programmers the ability to deal with untyped data easily. In particular, they do not support the ability to cast between untyped data and language-level types. We describe a powerful, type-safe cast operator that helps programmers write low-level systems codes in type-safe languages. We have implemented this operator in Modula-3 for the *SPIN* operating system, and we give specific examples of how we use it in *SPIN*.

KEY WORDS Language design Type safety Systems programming Casting

INTRODUCTION

An extensible system allows application-provided code to be linked in at runtime. The ability to extend a system at runtime enables a great deal of system flexibility, since the system can be dynamically customized to the

performance and functionality requirements of the application. Examples of extensible systems include Web browsers¹, operating systems², embedded systems³, and databases⁴. Such systems must be protected from imported code, which may be buggy or even malicious. Many modern extensible systems provide that protection through type-safe programming languages.

Type safety makes it possible for a system to provide strong guarantees of protection from imported code. In particular, a program cannot access arbitrary memory locations, and cannot use values in ways not prescribed by the language. Nevertheless, type-safe languages have not been designed to support efficient, low-level, “systems” code. For example, a network packet arrives as an uninterpreted array of bytes. In order to handle the packet, a network stack interprets the bytes in a packet as protocol headers, which contain fields of different lengths. For good performance, this interpretation should occur “in place”—that is, without copying⁵. In order to interpret untyped data in place, a cast operator is necessary. Unfortunately, type-safe languages do not traditionally provide non-trivial cast operators.

We describe a powerful, type-safe casting operator that makes type-safe languages more suitable for writing systems code. The design of the operator is based on the observation that if types have identical representations, the programmer can be allowed to cast between them. Our research has been done in the *SPIN* operating system², an extensible operating system that requires system extensions to be written in the type-safe subset of Modula-3⁶. We have implemented this cast operator in Modula-3 for *SPIN*, and we give several examples of how we use it to improve readability and performance. We discuss also how a safe cast operator could be introduced into other type-safe languages aimed at systems programming, such as Ada95⁷, Oberon⁸ or Java.

TYPE REPRESENTATIONS

Our cast operator allows a programmer to cast an lvalue (an expression that denotes a memory location) of some type to another type if the representations of the types allow doing so safely. For expository purposes we will not discuss the casting of rvalues, which is subsumed by lvalue casting. Casting makes sense if the types involved are *visible types*. A type is visible if it is non-abstract, i.e., its representation is exposed to its clients. In general,

abstract types cannot be cast, because their representations are unknown.

We can cast an lvalue from a visible type T_1 to another visible type T_2 when all of the following conditions hold:

- Every bit pattern that represents a legal value in T_1 represents a legal value in T_2 .
- If the lvalue is writable, every bit pattern that represents a legal value in T_2 must represent a legal value in T_1 .
- The use of the lvalue must satisfy the constraints of the underlying architecture. For example, we disallow casts between floating point and integer lvalues. Such casts might not preserve sharing if the lvalues were assigned to registers because the integer and floating point register sets are disjoint.
- The use of the lvalue must satisfy the code generation constraints of the compiler and architecture. For example, the alignment of the lvalue may have to satisfy the alignment required for T_2 .

Reasoning about the bit representation of types is straightforward (although compiler- and language-dependent):

- For a base non-pointer type, the set of legal bit representations is defined by the language implementation.
- For a pointer type, the set of legal representations is distinct from all other types.
- For a record or array type, the set of legal representations is a combination of the bit representations of the types of the fields (which must include any padding). This definition assumes that the constituents of a record or array as well as any padding are laid out according to a well-known, deterministic algorithm (as defined by the language or compiler).

Our casting operator is type-safe for two reasons. First, the conditions for legal casting guarantee that our cast does not create any illegal representations. Second, there is no harm in allowing application-provided code to “create” new instances of a visible type T via casting. Since T is a visible type, a client can allocate and modify its own instances the type anyway, so no protection is lost.

For the sake of exposition, we have assumed that the lvalue has the same size as T_2 . Nevertheless, the lvalue can be larger than T_2 , in which case it is effectively truncated by the cast. (Our notion of representation equivalence can be easily modified to account for truncation.) This feature is useful for looking at network packets, where networking code needs to examine only the first few bytes of a packet. For a dynamically sized lvalue,

such as an array, the size of the lvalue must be checked at runtime to guarantee that the lvalue is at least as large as T_2 .

EXAMPLE

We have implemented this cast operator, which we call **VIEW**, in Modula-3⁶. Modula-3 is a type-safe systems programming language that we have used to build the *SPIN* operating system, which runs on Alpha⁹ and x86 processors¹⁰. *SPIN* and its extensions are written almost entirely in Modula-3; **VIEW** lets us efficiently implement the code that interprets otherwise unstructured and untyped data, such as bytes coming from I/O devices. **VIEW** also enables safe interaction, without copying, between Modula-3 code and the portions of *SPIN* (such as the device drivers) that are written in C or assembly language.

The code in Figure 1 illustrates the difference between code that uses **VIEW** and code that does not. The figure contains three possible versions of packet filters. The actual packet filter used in the *SPIN* network subsystem¹¹ would use **VIEW**: that is, it would be the procedure called `ViewFilter`.

The first filter, `CopyFilter`, is representative of a naive implementation of the packet filter in a type-safe language. In order to manipulate the packet as a packet header record, the filter must invoke a system-provided routine that copies the packet (as an array of bytes) into a record. It is important that to note that the system must provide the trusted routine `Ip.ConvertByCopyingPacket`, which performs the copy. The routine must be trusted, because it is not type-safe. If we required the use of `CopyFilter`, we would have to ensure that the system provided sufficient copying routines (in order to cover all anticipated extensions), which would be difficult.

The second filter, `ByteFilter`, is representative of a second, more efficient implementation of the packet filter in a type-safe language that does not support **VIEW**. In order to avoid copying the packet, the filter explicitly manipulates the bytes of the packet. The first problem with this code is that it is hard to write, read, and maintain. Semantic information about the packet fields is not present, because the packet is an array of bytes. The second problem is that it could be inefficient on some architectures. For example, the Alpha does not provide instructions that operate on bytes: without a fairly smart compiler, the code that is generated could contain a number of useless mask operations.

The third filter, `ViewFilter`, is the implementation of the packet filter using **VIEW**. Inside the body of the **WITH** statement, the variable `IpHeader` is a typed alias for the packet’s header. **VIEW** enables programmers to write code that is simpler to understand and that executes more efficiently than using explicit byte manipulations.

```

MODULE Ip;

IMPORT Word;

(* IP address of www-spin.cs.washington.edu *)
CONST SourceAddr = 16_805f02DE;

(* type-safe, readable, but has a copy *)
PROCEDURE CopyFilter(READONLY m: ARRAY [0..255] OF Byte) : BOOLEAN =
  VAR ipHeader:Ip.Header := Ip.ConvertByCopyingPacket (m); (* calls bcopy *)
  BEGIN
    RETURN ipHeader.protocol = Ip.protocol.UDP AND
           ipHeader.version = Ip.version4 AND
           ipHeader.sourceAddr = SourceAddr;
  END CopyFilter;

(* type-safe, but unreadable; inefficient on machines without byte operations *)
PROCEDURE ByteFilter(READONLY m: ARRAY [0..255] OF Byte) : BOOLEAN =
  BEGIN
    RETURN a[9] = Ip.protocol.UDP AND
           Word.And (m.packet[0], 16_F) = Ip.version4 AND
           Word.Or(Word.LeftShift(m.packet[15], 24),
                  Word.Or(Word.LeftShift(m.packet[14], 16),
                          Word.Or(Word.LeftShift(m.packet[13], 8),
                                  m.packet[12]))) = SourceAddr;
  END ByteFilter;

(* type-safe, readable, and efficient *)
PROCEDURE ViewFilter(READONLY m: ARRAY [0..255] OF Byte) : BOOLEAN =
  BEGIN
    WITH ipHeader = VIEW(m, Ip.T) DO (* no copying *)
      RETURN ipHeader.protocol = Ip.protocol.UDP AND
             ipHeader.version = Ip.version4 AND
             ipHeader.sourceAddr = SourceAddr;
    END;
  END ViewFilter;

BEGIN
END Ip.

```

Figure 1. Implementation of packet filters using **VIEW** and byte operations. The filters accept unfragmented UDP/IP packets with a particular source address. The **WITH** operator creates an alias to the `ipHeader` for the **VIEW** expression.

Table I illustrates the low-level performance of using **VIEW**, measured on a DEC Alpha AXP 3000/700. Instructions were hand-counted, and cycles were measured using the Alpha cycle counter. The cycle measurements were taken with a warm data cache (the packet being examined), and a cold instruction cache. The first three filters are the ones shown in Figure 1, and were compiled with our version of the DEC SRC Modula-3 compiler. The filter `Cfilter` is the equivalent filter written in C, and compiled using Digital `cc`. Note that `Cfilter` is not type-safe. The performance of `Cfilter` merely serves as a reference point for our performance

comparison.

`CopyFilter` is the slowest filter, because it incurs the cost of copying the packet header. `ByteFilter` is significantly faster, but it is harder to write, read, and maintain. `ViewFilter` is as easy to read as `CopyFilter`, and its performance is actually better than `ByteFilter`. The performance improvement is due to the fact that the Alpha architecture does not support byte instructions, and the compiler cannot recognize that no shifts and masks are necessary.

	instructions	time (in cycles)
CopyFilter	170	250
ByteFilter	37	71
ViewFilter	25	65
CFilter	15	53

Table I. Comparison of different packet filters. on a DEC Alpha AXP 3000/700 running at 225 MHz, with 160MB of memory. The first three filters are written in Modula-3, and are type-safe. Cfilter is written in C, and is not type-safe; it serves as a reference point.

Out of the three type-safe filters, the performance of ViewFilter is the closest to that of Cfilter. Cfilter is slightly faster than ViewFilter for three reasons:

- **VIEW** performs an alignment check, which takes two instructions. A C cast does not perform any checks.
On a system where the operating system, compiler, or hardware supports unaligned memory operations, the alignment check is unnecessary. For example, the x86 architecture directly supports unaligned loads and stores, so, the x86 version of our Modula-3 compiler does not generate an alignment check.
- Because of the alignment check, the Modula-3 compiler generates a potential call to a generic error-reporting procedure. Ignoring this procedure, the filters are all leaf procedures. The C compiler treats Cfilter as a leaf, which saves five instructions.
- The DEC C compiler generates better code, which saves three instructions.

Over half of the extra instructions come from the alignment check. It should be noted that we have also added tighter control over the alignment of types in Modula-3 for *SPIN*, so this alignment check can be eliminated when the device driver only returns correctly aligned packets.

DISCUSSION

Since 1995 we have used Modula-3⁶ to develop operating system services for the *SPIN* operating system. To date we have implemented a variety of operating system extensions in Modula-3, such as user-level threads, a complete TCP/IP protocol stack, transaction services for databases, a log-structure file system, NFS and HTTP servers, and a Unix emulation library that is binary compatible with Digital UNIX and FreeBSD. We have found Modula-3 to be an effective type safe programming language to develop a high-performance, modular system.

The type-safe cast operator is quite useful to other operating system services that operate on data created outside of the language. The most common use, similar to its use in the network stack, is to interpret I/O data as a record. Several examples of this type of use are as follows:

- **VIEW** is used to interpret file data as COFF object file headers¹².
- **VIEW** is used to interpret disk data as file system structures, such as inodes.
- **VIEW** is used to interpret blocks of data as UNIX socket structures.
- **VIEW** is used to interpret untyped system call arguments as more structured types.

The code for these uses looks similar to the use in the network stack.

VIEW is also used in *SPIN* for reasons other than interpreting external data such as to overcome interface mismatches within the *SPIN* system. For example, **VIEW** is used to translate data between formats that are logically equivalent, such as integers and arrays of bytes. Certain interfaces are written to use integers, and others are written to use arrays of bytes. Both sets of interfaces are correct by themselves, and **VIEW** allows them to be used together without having to rewrite them.

Finally, **VIEW** is used for some minor operations, such as converting pointers to integers for the sake of error reporting or debugging. Readonly pointers can be safely cast to integers using **VIEW**, as described earlier.

The ability to safely cast with **VIEW** supports other low-level data manipulations, which are not directly used in *SPIN* currently. First, a programmer can use casting to implement record subtyping by hand. Such a feature is useful in languages where record types are insufficiently flexible. For example, **VIEW** can be used to truncate a network packet to just its header, so that the header can be passed to a networking routine. Second, a programmer can also use casting to get around restrictive type equivalence rules. For example, Modula-3 does not allow record types whose field names are different to be type-equivalent. **VIEW** can be used to treat a value of one such type as an instance of the other type.

In general, type-safe casting is useful in languages that have a type system with a rich set of lvalue types. For example, Ada95 and Oberon are systems languages that have a rich set of lvalue types and both lack a type-safe cast operator. These languages would benefit from the addition of a type-safe cast operator. In pure object-oriented languages without “manifest” records and arrays, the ability to cast would not be as useful. In languages such as Java¹ and Theta¹³ the only lvalue types are pointers (object instances), and the integral and floating-point types. **VIEW** would not be that useful in these languages, because they provide neither visible records nor arrays such as C provides. (**VIEW** cannot be

used to cast between sibling object types, because such casts would break the type system.) From our experience in building an operating system, such high-level type systems are unacceptable, because of the need to directly access I/O data as well as C-style data structures in a mixed language environment. The need for such visible types is proven by the need for native methods in Java: Modula-3 can be efficiently implemented in itself, whereas Java must resort to C code for its low level runtime and I/O.

Finally, we note that any use of our cast operator results in unportable (but type-safe) code. For example, the endianness of a machine affects the meaning of a cast. Nevertheless, the use of our cast operator is no more unportable than the use of cast operations in C. Given that the purpose of casting is to allow us to write low-level systems code, unportability is not a defect.

RELATED WORK

Mechanisms similar to **VIEW** have been provided in other languages. The Modula-2+ ϵ ¹⁴ language allows the Modula-2 **LOOPHOLE** construct to be used in restricted ways within safe modules. The safe **LOOPHOLE** in Modula-2+ ϵ is much weaker than **VIEW**, as it only can be used to cast rvalues (expressions that denote values).

The Fox project at CMU has been investigating the construction of system software in ML. To support functionality similar to that provided by **VIEW**, the Fox group has defined a set of generic interfaces for sequence types¹⁵. These interfaces support operations such as iteration over homogeneous aggregate types (lists and arrays). For example, an array of bytes can be traversed as an array of words. These sequence interfaces support both little-endian and big-endian access to sequences, whereas **VIEW** does not. **VIEW** supports more of the manipulations that we desire, though; it allows byte arrays to be interpreted as heterogeneous aggregates, such as record types. Such a feature is useful in networking, where packet headers contain fields of different sizes. The Fox sequence interfaces only allow “casts” between homogeneous aggregates.

CONCLUSIONS

We have described a powerful type-safe cast operator. This operator allows programmers to write efficient and clear low-level systems programs, and we have used it extensively in *SPIN*. Although we have motivated our cast operator in terms of extensibility, it is useful in any

type-safe language where the ability to write low-level systems programs is desired. Type-safe casting supports various kinds of data manipulation that could otherwise be performed only by going outside of the language. We have given a detailed example of how the cast operator is used in the network stack in *SPIN*, as well as a description of how we have used it elsewhere in *SPIN*.

REFERENCES

1. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, The Java Series, Addison-Wesley, 1996.
2. B.N. Bershad, S. Savage, P. Paradyak, E.G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers, ‘Extensibility, safety and performance in the *SPIN* operating system’, *15th SOSP*, Copper Mountain, CO, December 1995, pp. 267–284.
3. Inferno: la commedia interattiva. <http://inferno.lucent.com/inferno/infernosum.html>, 1996. A brief overview of the Inferno OS.
4. B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shriram, ‘Safe and efficient sharing of persistent objects in Thor’, *SIGMOD '96*, Montreal, Canada, June 1996.
5. D. D. Clark and D. L. Tennenhouse, ‘Architectural considerations for a new generation of protocols’, *SIGCOMM '90*, September 1990.
6. *Systems Programming with Modula-3*, ed., G. Nelson, Series in Innovative Technology, Prentice Hall, Englewood Cliffs, NJ, 1991.
7. International Organization for Standardization, *Ada 95 Reference Manual. The Language. The Standard Libraries*, January 1995. ANSI/ISO/IEC-8652:1995.
8. H. Mössenbock and N. Wirth, ‘The programming language oberon-2’, *Structured Programming*, **12**, (4), 179–195, (1991).
9. R.L. Sites and R.T. Witek, *Alpha AXP Architecture Reference Manual*, Digital Press, 2nd edition, October 1995.
10. Intel Corporation, Mt. Prospect, IL, *Pentium Pro Family Developer's Manual*, 1997.
11. M. Fiuczynski and B.N. Bershad, ‘An extensible protocol architecture for application-specific networking’, *Winter USENIX '96*, San Diego, CA, January 1996, pp. 55–64.
12. G. R. Gircys, *Understanding and Using COFF*, Nutshell Series, O'Reilly & Associates, Inc., Sebastopol, CA, November 1988.
13. B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A.C. Myers, ‘Theta reference manual’, Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, (February 1995). <http://clef.lcs.mit.edu/papers/thetaref/index.html>.
14. Digital Equipment Corporation, *Modula-2+ ϵ Language Specification*, March 20, 1991.
15. E.S. Biagioni, ‘Sequence types for functional languages’, Technical Report CMU-CS-95-180, CMU SCS, Pittsburgh, PA, (August 1995).