41st EuroForth Conference

Hamburg, Germany
September 12-14, 2025

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 41st EuroForth finds us in Hamburg; in 2024 it was held in Newcastle upon Tyne, and in 2023 in Rome. Information on earlier conferences can be found at the EuroForth home page (`http://www.euroforth.org/`).

Since 1994, EuroForth has a refereed and a non-refereed track. This year there have been no submissions to the refereed track. Since 2006, there have been 32 submissions, 23 accepts, 72% acceptance rate.

Several papers were submitted to the non-refereed track in time to be included in the at-conference proceedings. Late papers as well as those slides that were submitted to the editor are included in these post-conference proceedings. The proceedings website `http://www.euroforth.org/ef25/papers/` also contains links to videos of the presentations. I thank the authors for their papers and slides. In addition to the papers and presentation handouts available before the conference, these online proceedings also contain papers and presentation handouts that were provided at or after the conference. Also, some of the papers included in the printed proceedings were updated for these online proceedings. I thank the authors for their papers and slide handouts.

You can find these proceedings, as well as the individual papers and (when they become available) slides and links to the presentation videos on `http://www.euroforth.org/ef25/papers/`.

Workshops and social events complement the program. This year's EuroForth was organized by Ulrich Hoffmann.

Anton Ertl

Program committee

M. Anton Ertl, TU Wien (chair)
Marcel Hendrix, Eindhoven University of Technology
Jaanus Pöial, Tallinn University of Technology
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart
Reuben Thomas

Contents

Non-Refereed Papers

Bill Stoddart: Investigating Goodstein Sequences 5

Nick J. Nelson: Forth 2025 12

Nick J. Nelson: Improvements to enumeration 24

M. Anton Ertl and Bernd Paysan: Code-Copying Compilation in Production — An Experience Report 29

Late Non-Refereed Papers

Stephen Pelc: VFX Forth for AA64 Linux 46

Presentation Slides

Ulrich Hoffmann: Blending Forth - mixing other Languages and Forth 51

M. Anton Ertl: What is a character? UTF-8, Unicode, and the Xchar wordset 61

Investigating Goodstein Sequences

Bill Stoddart

September 12, 2025

Abstract

The weak and strong Goodstein theorems are examples of strongly counter intuitive results concerning certain integer sequences that typically grow very rapidly but eventually converge to zero. In this paper we describe the weak Goodstein sequences and aim provide the reader with an *intuitive* understanding of why the sequences converge. After that, although the paper remains descriptive, it takes a mathematical turn. We introduce transfinite ordinal numbers and demonstrate a decreasing sequence which bounds a weak Goodstein sequence above and terminates in zero. We consider the extraordinary behaviour of the strong Goodstein sequence and show how to construct corresponding decreasing sequences of transfinite ordinals. We call for help from an AI model to do the heavy lifting we require for our algebraic manipulations.

1 Introduction

A weak Goodstein sequence is constructed by choosing any number and and expressing it in base 2. Successive numbers are formed by reinterpreting the expression of the current number after incrementing the base, then subtracting 1. The Forth program G performs these steps, taking as input the first number in the current sequence. Starting with 266 the first two steps are:

266 G
$100001010_2 \ = \ 266_{10}$
$100001010_3 - 1 = 100001002_3 = 6590_{10}$

At each step we increase the base, in this case from 2 to 3. Reinterpreting the same string of characters in the new base gives a value which has increased from 266 to 6591. We then subtract 1.

Here are the following three steps:

$100001002_4 - 1 = 100001001_4 = 65601_{10}$
$100001001_5 - 1 = 100001000_5 = 390750_{10}$
$100001000_6 - 1 = 100000555_6 = 1679831_{10}$

Increasing the base and reinterpreting the same string of figures has an enormous effect, again which the effect of subtracting 1 seems relatively insignificant. So why do such sequences converge to zero?

One initial hint can be seen in the above example. Although successive terms grow rapidly in the above example their representations do not grow in size.

2 A small example

We take the example if $4 \ = \ 100_2$.

Reinterpreting the string 100 in base 3 gives us $100_3 \ = \ 9_{10}$.

Now subtracting 1 we see $100_3 \ - 1 \ = \ 22_3$ Subtracting 1 has required a carry and in this case has reduced the size of the representation from 3 figures to 2.

Let's follow the process and see if the 2 figures will eventually reduce to 1.

$$\begin{array}{l}
4 \ G \\
100_2 \ = \ 4_{10} \\
100_3 - 1 = 22_3 = 8_{10} \\
22_4 - 1 = 21_4 = 9_{10} \\
21_5 - 1 = 20_5 = 10_{10} \\
20_6 - 1 = 15_6 = 11_{10} \\
15_7 - 1 = 14_7 = 11_{10} \\
14_8 - 1 = 13_8 = 11_{10} \\
13_9 - 1 = 12_9 = 11_{10} \\
12_{10} - 1 = 11_{10} = 11_{10} \\
11_{11} - 1 = 10_{11} = 11_{10} \\
10_{12} - 1 = B_{12} = 11_{10}
\end{array}$$

When we arrive at base 6 our representation has a leading 1. At this point increasing the base increments the sequence value by 1 and subtracting 1 reduced it by 1 so the sequence values are identical until we arrive at base 12.

At this point our representation is $10_{12} - 1 = B_{12}$ and we have reduced our representation to a single figure. With a single figure representation increasing the base has no effect, so our sequence terms decrease by 1 at each step as follows:

$$\begin{array}{l}
10_{12} - 1 = B_{12} = 11_{10} \\
B_{13} - 1 = A_{13} = 10_{10} \\
A_{14} - 1 = 9_{14} = 9_{10} \\
9_{15} - 1 = 8_{15} = 8_{10} \\
8_{16} - 1 = 7_{16} = 7_{10} \\
7_{17} - 1 = 6_{17} = 6_{10} \\
6_{18} - 1 = 5_{18} = 5_{10} \\
5_{19} - 1 = 4_{19} = 4_{10} \\
4_{20} - 1 = 3_{20} = 3_{10} \\
3_{21} - 1 = 2_{21} = 2_{10} \\
2_{22} - 1 = 1_{22} = 1_{10} \\
1_{23} - 1 = 0_{23} = 0_{10}
\end{array}$$

3 Potential and Achievement

Following the above discussion we wonder if we might be able to introduce some definitions that that in some sense can capture the eventual reduction of the length of our representation is a more finely calibrated way. With this in mind we introduce Potential and Achievement.

Returning to our first example of $266 = 100001010_2$ we are starting with a number which requires 9 bits for its representation. The maximum value we can represent in binary with 9 characters at our disposal is 512-1 = 511. We call this the potential of a 9 bit binary number. The value 100001010_2 achieves $266/511 = 0.5205$ of its potential, and we will say it has an achievement of .5205. The maximum value we can represent in 9 places with a base 3 representation is $3^9 - 1 = 19628$. The second value in our sequence is 6590, so its achievement is $6590/19628 = 0.3450$.

This pattern continues, so that as the values in the sequence initially increase before eventually decreasing, the achievements of the terms is always decreasing. Since an achievement of zero is associated with a number which is zero, if we could prove that the achievements converge to zero, this would prove the sequence converges to zero. However, proving the convergence of a sequence of real values terms to zero is not in general easy to do, wheras a decreasing sequence of positive whole numbers willalways decrease to zero. Could we perhaps produce a decreasing sequence of positive whole numbers to act as upper bounds to our sequence? With a sequence that increases to rapidly this would seem difficult, but we will do it by admitting transfinite numbers.

4 Transfinite Numbers - defining numbers with sets

The traditional proof of convergence for Goodstein sequences uses Cantors hierarchy of "ordinal numbers". We define numbers in terms of sets, as follows.

$$\begin{array}{lll} 0 \;\widehat{=}\; \{\,\} & & \text{zero will be modelled by the empty set.} \\ 1 \;\widehat{=}\; \{0\} & & \\ 2 \;\widehat{=}\; \{0,1\} & & \\ \ldots & & \\ n \;\widehat{=}\; \{0,1,2,\ ..\ n-1\} & & n \text{ is defined as the set of all numbers } \textit{less than } n. \end{array}$$

Then if numbers $a, b,\ a < b$ when $a \subseteq b$

The smallest set that is bigger than all the finite numbers is referred to as ω, and is defined as follows:

$\omega \;\widehat{=}\; \{0, 1, 2,\ \ldots\ \}$.

This is our first transfinite number. We can define its successor as follows:

$\omega + 1 \;\widehat{=}\; \{\omega, 0, 1, 2.....\}$

We can continue in this way defining a hierarchy of transfinite numbers

$\omega,\ \omega+1,\ \omega+2,\ ...\ 2\omega,\ 2\omega+1,\ ,,,\ 3\omega\ ,\ ...\ \omega^2,\ ...\ \omega^3,\ ...\omega^\omega,\ ....$

Note that although ω has a successor, it has no predecessor. $\omega - 1$ is undefined. It shares this property with other "limit ordinals", e.g. 2ω, ω^2, ω^ω etc.

5 Bounding the terms of a Goodstein sequence using ordinal numbers

The terms of any Goodstein sequence can be bounded above by a decreasing sequence of ordinals. A well known theorem states that any decreasing sequence of ordinals terminates in zero, and we can call on this theorem to show that any Goodstein sequence terminates in zero.

We give an example to show such a sequence of ordinals is created, and how it decreases.

Base	Term	Transfinite bound
2	$100_2\ =\ 2^2$	ω^2
3	$3^2 - 1\ =\ 2\times 3+2$	$2\omega+2$
4	$2\times 4+2-1\ =\ 2\times 4+1$	$2\omega+1$
5	$2\times 5+1-1\ =\ 2\times 5$	2ω
6	$2\times 6-1\ =\ 6+5$	$\omega+5$
7	$7+5-1\ =\ 7+4$	$\omega+4$
8	$8+4-1\ =\ 8+3$	$\omega+3$
9	$9+3-1\ =\ 9+2$	$\omega+2$
10	$10+2-1\ =\ 10+1$	$\omega+1$
11	$11+1-1\ =\ 11$	ω
12	$12-1\ =\ 11$	11
13	$11-1 =\ 10$	
...		

We have shown the sequence up to the point where the value of the current term is represented by a single figure. From this point onwards any increase in the base has no effect on the next value, so terms decrease by 1 at each step until 0 is reached.

6 The Strong Goodstein Sequence

Strong Goodstein sequences are specifically designed to grow very rapidly. The strong Goodstein sequence starting at 266 has the following values for its first 5 terms, yet eventually converges to zero.

$266,\ 4.4\ \times\ 10^{36},\ 3.2\ \times\ 10^{616},\ 2.5\ \times\ 10^{10972}$

For perspective one can bear in mind that that the number of atoms in the observable universe is commonly estimated to be in the region of 10^{80}

A strong Goodstein sequence starts with a binary number expressed in "hereditary base notation". This notation restricts us to describing base n numbers using exponentiation on n and addition of numbers 1 to n-1 to fill in intermediate values. For example in base 2:

$$\begin{aligned}
&1 \;=\; 1,\; 2 \;=\; 2,\; 3 \;=\; 2+1,\; 4 \;=\; 2^2,\; 5 \;=\; 2^2+1,\; 6 \;=\; 2^2+2,\\
&7 \;=\; 2^2+2+1,\; 8 \;=\; 2^{2+1},\; 16 \;=\; 2^{2^2}\\
&31 \;=\; 2^{2^2}+2^{2+1}+2^2+2+1
\end{aligned}$$

The proof of convergence of the sequence constructs a corresponding decreasing sequence of transfinite terms obtained by replacing the base by ω. As an example we will take the sequence whose first term is 16, which is expressed in hereditary base notation as 2^{2^2} The initial transfinite term is obtained by replacing each 2 in the base 2 term with ω, giving ω^{ω^ω}. The next term in the numeric sequence is obtained by replacing each 2 in the hereditary base representation by 3 and subtracting 1, giving $3^{3^3} \;-\; 1$. However, this is not in hereditary base 3 form. The next transfinite term is given by expressing the numeric term in hereditary base 3 and replacing each occurrence of the base 3 by ω.

7 Help from an AI model

Converting $3^{3^3} \;-\; 1$ to hereditary base 3 and producing the related transfinite ordinal term requires non-standard and relatively heavy algebraic manipulation, so I was curious to see if it would be useful to enlist the help of an AI agent, in this case chatGPT 5, which had just been released and was supposed to be capable of PhD level maths. After initial incorrect attempts in which it did not interpreted hereditary base notation correctly, and I had to remind it that $\omega - 1$ was undefined, and after being given the hint that since $3^{3^{3-1}}$ has the form $x^3 - 1$, we can use the identity $x^3 - 1 \;=\; (x-1)(x^2+x+1)$, it did produce the complex formulae used in the rest of this section.

$$\begin{aligned}
&3^{3^3} - 1 \;=\\
&2\times 3^{2\times 3^2\,+\,2\times 3\,+\,2} \;+\; 2\times 3^{2\times 3^2\,+\,2\times 3\,+\,1} \;+\; 2\times 3^{2\times 3^2\,+\,2\times 3} \;+\\
&2\times 3^{2\times 3^2\,+\,3\,+\,2} \;+\; 2\times 3^{2\times 3^2\,+\,3\,+\,1} \;+\; 2\times 3^{2\times 3^2\,+\,3} \;+\\
&2\times 3^{2\times 3^2\,+\,2} \;+\; 2\times 3^{2\times 3^2\,+\,1} \;+\; 2\times 3^{2\times 3^2} \;+\\
&2\times 3^{3^2\,+\,2\times 3\,+\,2} \;+\; 2\times 3^{3^2\,+\,2\times 3\,+\,1} \;+\; 2\times 3^{3^2\,+\,2\times 3} \;+\\
&2\times 3^{3^2\,+\,3\,+\,2} \;+\; 2\times 3^{3^2\,+\,3\,+\,1} \;+\; 2\times 3^{3^2\,+\,3} \;+\\
&2\times 3^{3^2\,+\,2} \;+\; 2\times 3^{3^2\,+\,1} \;+\; 2\times 3^{3^2} \;+\\
&2\times 3^{2\times 3\,+\,2} \;+\; 2\times 3^{2\times 3\,+\,1} \;+\; 2\times 3^{2\times 3} \;+\\
&2\times 3^{3\,+\,2} \;+\; 2\times 3^{3\,+\,1} \;+\; 2\times 3^{3} \;+\\
&2\times 3^2 \;+\; 2\times 3 \;+\; 2
\end{aligned}$$

Note, we don't go directly to calculating the *value* of the expression. We keep it in this form, in which 3 is the current number base, so that we can obtain the corresponding transfinite term by replacing each 3 by ω.

$$\begin{array}{l}
2\omega^{2\omega^2 + 2\omega + 2} + 2\omega^{2\omega^2 + 2\omega + 1} + 2\omega^{2\omega^2 + 2\omega} + \\
2\omega^{2\omega^2 + \omega + 2} + 2\omega^{2\omega^2 + \omega + 1} + 2\omega^{2\omega^2 + \omega} + \\
2\omega^{2\omega^2 + 2} + 2\omega^{2\omega^2 + 1} + 2\omega^{2\omega^2} + \\
2\omega^{\omega^2 + 2\omega + 2} + 2\omega^{\omega^2 + 2\omega + 1} + 2\omega^{\omega^2 + 2\omega} + \\
2\omega^{\omega^2 + \omega + 2} + 2\omega^{\omega^2 + \omega + 1} + 2\omega^{\omega^2 + \omega} + \\
2\omega^{\omega^2 + 2} + 2\omega^{\omega^2 + 1} + 2\omega^{\omega^2} + \\
2\omega^{2\omega + 2} + 2\omega^{2\omega + 1} + 2\omega^{2\omega} + \\
2\omega^{\omega + 2} + 2\omega^{\omega + 1} + 2\omega^{\omega} + \\
2\omega^2 + 2\omega + 2
\end{array}$$

The important point about this expression is that it represents a transfinite number less than ω^{ω^ω}. We can see this by comparing ω^{ω^ω} to the highest order term in the expression.

A reader might conceivable wonder why go to such bother. Can't we just use $\omega^{\omega^\omega} - 1$ as our next transfinite term? The reason is that ω^{ω^ω} is a limit ordinal and has no predecessor, so $\omega^{\omega^\omega} - 1$ is undefined.

8 Background and Related Work

Reuben Goodstein introduced his sequences in the paper "On the restricted ordinal theorem", Journal of Symbolic Logic, Vol. 9, No. 2 (June 1944), pp. 33–41. The sequences were specifically designed to provide an application for transfinite ordinals, and hereditary base notation was specifically introduced for describing the strong sequences. In L. Kirby and J. Paris, "Accessible independence results for Peano Arithmetic," Bulletin of the London Mathematical Society 14 (1982), 285–293, we find a proof that convergence of the strong Goodstein sequence cannot be proved in the usual formalisation of arithmetic using Peano's axioms. This result is sometimes cited as an illustrative example for Gödel's incompleteness theorem, which states that all non-trivial mathematical theories must include results which are true but unprovable *within the theory.* An unpublished paper of Cansell and Abrial represents the strong Goodstein sequences as finite trees, and shows proof of their convergence can be deduced from the properties of these finite trees.

9 Conclusions

We've looked at Goodstein Sequences and tried to give some understanding of why these converge to zero despite initially growing so rapidly. The weak Goodstein sequence grows by interpreting the same string of figures in a new base, incremented by 1 at each step, then subtracts 1. Our first clue as to why the -1 steps wins out over the base increase is to notice that the number of figures used in the representation of each step does not increase. This allows the -1 step to erode successive values until the value of the current term is expressed as a single figure. At this point the base increase has no further effect and the value of successive terms decreases by 1. Another clue is given by why value each term achieves in comparison with its maximum in the current base. The "achievement" of successive terms diminishes.

The classic convergence proofs forGoodstein sequences use transfinite ordinals. We give a short introduction to ordinal numbers, with finite numbers defined in terms of finite sets, and show how this can be extended to introduce "transfinite" numbers $\omega,\ \omega+1, ...$ which are represented by infinite sets. We show how each weak Goodstein sequence is bounded above by a companion sequence whose initial terms are transfinite ordinals and whose terms decrease, thus, by a well known property of ordinals, inevitable arrive at zero.

Terms in the companion sequence are obtained by taking the expression of each term in the first sequence, written in terms of the current base, and replacing each occurrence of the base value by ω.

The same method is used with the strong Goodstein sequence, in which terms are written using hereditary base notation In this notation, the coefficients of a representation are also expressed in terms of the current base, greatly increasing the effect of a base increment and generating some of the largest numbers encountered in number theory. However we can still use the trick of producing a decreasing companion sequence of upper bounds. Here we limited ourselves to showing how one such term was generated, using an AI model to help us with the algebraic manipulations.

Forth 2025

Abstract

Forth is stuck in a rut. Much energy seems to be spent in making tiny refinements to a language specification that is 25 years old. In this paper, I shall propose some bold and radical changes, with the intention of returning Forth to its proper place as a useful and modern language.

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. Introduction

It often feels like Forth is moribund! In the last few years, at this conference, I seem to be in a minority of delegates from commercial companies that use Forth in their main products. Yet, Forth still has huge advantages, and in the last few years at EuroForth, I have tried to highlight extraordinarily useful techniques that are possible in Forth and *are not possible in any other language that I know of.*

One has to ask, why do more people not use Forth, given these advantages? Here are some possible answers:

a) These advantages are not stressed, in online descriptions of Forth. Instead, one sees long descriptions of how antiquated the language is, and how quirky, and by inference how it is only chosen nowadays (rarely) for "niche" applications.

b) If someone, who is assessing which new language to choose, accidentally comes across the Forth standards website, they will ask (after a near death experience due to terminal boredom) "yes, but what is it for, what can it actually do, that others cannot?" - and no answer is given.

c) Forth really does have some antiquated absurdities, the reasons for which are lost in the mists of time.

d) Forth does not initially *appear* to have many features that are essential for modern programming.

2. What are the true advantages of Forth?

a) You can do things *during compilation*. This includes quite complicated things like querying databases. This feature opens up a whole realm of extraordinary possibilities, some of which I have attempted to demonstrate at many previous conferences.

b) In fact, you can do *anything* in Forth. Although there may be guidelines, you are not prevented from doing anything you like. (Of course, this also enables an incautious programmer to get into deep trouble.)

c) Forth can be completely freely formatted. This *enables and encourages* you to write concise, highly readable and easily maintainable code. (Of course, it also enables you to write complete gibberish, if you really want to.)

d) Forth is interactive, and you can easily arrange for this to continue *even while your application is running*.

e) The edit-compile-execute-test-debug cycle is extraordinarily quick and efficient in Forth. This is mainly because everything except the edit is done within Forth itself. The old adage "Forth is its own compiler" is still just as true today.

f) If there's some Forth word you don't like, you can always redefine it.
See section 5 below!

3. What previous advantages of Forth no longer apply?

In the past, Forth was often described as being fast, compact, and so simple that anyone could write their own compiler in a day.

It is true that Forth is still fast and compact. However, to achieve speed when targetting a modern and highly complex CPU, you need an optimising compiler, which is not built in a day.

As regards compactness, who cares any more? If your program won't fit, spend 5 Euros on some more memory.

4. What do we need to do?

- Get rid of the bad bits.
- Enhance the good bits.
- Add the missing bits.

5. The bad bits - WITHIN

I have never known a word, in any language, that has caused so much trouble to so many programmers.

```
3 1 3 WITHIN BOOL. False  ok
```

The worst thing about it is that because this affects a boundary condition, a mistake may not be seen until an application has been running for weeks.

We got to this state because of incorrect mathematics, which assumed that the input parameters were real numbers. But they are not - they may only be integers. The correct mathematics is to ask whether the integer 3 is within the set {1,2,3}.

I guess it's not possible to completely get rid of WITHIN, but one could at least move it to the "Optional Badwords" wordset! Then we could introduce a new core word - perhaps MEMBER - which is inclusive.

At the very top of every build file, we have to redefine WITHIN. It occurs 1094 times in our main application.

6. The bad bits - CASE

It's the default clause of a CASE construction that causes so many programming errors. Again, the mistake might not be seen for a long time. The perfectly simple solution is to keep the index on the return stack instead of the number stack.

At the very top of every build file, we have to redefine CASE and all its other words. A CASE construction occurs 830 times in our main application.

7. The DO...LOOP conundrum

Observations

a) There are 1612 DO...LOOP constructions in our main application.

b) We currently have a rule that if either of the two input parameters is not a constant, then we always use ?DO instead of DO. We have 1018 ?DOs and 594 DOs.

c) Earlier on in our development process, we defaulted to 0-based indexing (computer friendly). About two years ago, we changed to 1-based indexing (human friendly). We did not modify old 0-based code.

d) We have 1281 instances of either 0 DO or 0 ?DO.
We have 329 instances of either 1 DO or 1 ?DO.
In only two cases out of 1612 do we use anything other than 0 or 1 as the second parameter.

e) A disadvantage of using 1-based indexing is that the first parameter for DO frequently requires 1+. Forgetting to do the 1+ is a common cause of programming errors.

f) There are only 3 instances of +LOOP.

Conclusions

a) Move DO, ?DO, LOOP and +LOOP to an optional wordset.

b) Introduce a new and much simpler looping construct, perhaps
FOR (n---) ... NEXT which loops n times, but skips completely
if n < 1.

c) Retain the same return stack structure as a DO...LOOP, so that
I, J and LEAVE work as before.

This would satisfy 99.8% of our loop requirements, with the benefit of greater security and simplicity.

8. Values not variables

As far back as Euroforth 2000, I started advocating for the deprecation of VARIABLE, @ and !, and the promotion of VALUE. The reasoning for that may be found in my previous paper.

Modifiers (they should not be called operators) should be standardised, enumerated and extendable. Attempting to apply a modifier to an unmodifiable word should give a compilation error.

At present, our main application has 105 remaining VARIABLEs - these are mostly because we have not yet redefined the VFX chain functions, which currently require a VARIABLE root. By contrast, there are 1334 VALUEs.

The issue of thread local VARIABLEs or VALUEs will be discussed in section 13.

9. Add VINDEX, VMATIX, STRINDEX

What modern language could be without proper standardised support for arrays and matrices?

My Euroforth 2000 paper also proposed:
a) VINDEX for an array of CELLs
b) VMATRIX for a two dimensional matrix of CELLs
c) STRINDEX for an array of strings.
d) VFIELD and derivatives, for structures

It turned out that some very large VINDEXs and VMATRIXs were wasting a lot of space by using 64 bit cells when for example only byte values were required. We have therefore extended the concept to provide byte, word and int (32 bit) flavours.

By now, we can't imagine how we managed without them.

There are 210 VINDEXs, 28 VMATRIXs and 26 STRINDEXs in our main application. This excludes the byte, word and int flavours (there are for example 15 VBINDEXs), and the dynamically generated VALUEs, VINDEXs and STRINDEXs resulting from the replacement for the Windows registry settings, also described in a different 2020 paper.

I should add that I am not particularly happy with the naming of some of these words, and would be quite happy to change the names, provided the new name was shorter.

10. Add ENUM<<

How is it possible for a language to survive without a standardised enumeration function? I described a fully Forth faithful enumeration at Euroforth 2023, and some enhancements were described earlier at this conference. Surely, this should at least form a part of an optional word set?

11. Zstrings

It must be at least 30 years ago that I first started advocating a general move from counted strings to zero terminated strings. The original reason was that any serious application is likely to need to interact with the native operating system, and also external libraries with "C" interfaces, and, of course, Windows, Linux and C libraries all use zstrings. Back in the days of Windows 95, it was not certain which side the coin would fall, and the 32 bit Windows version of our main applications extensively used "bi-strings", that is zero terminated counted strings. By the time we moved to Linux, initially 32 bit then soon after 64 bit, it became clear that we were in a zero terminated world, and support for both cstrings and bi-strings was dropped.

As I wrote this, I searched for "zero terminated" in the Forth standard and got "NO RESULTS" - which gives the clear impression that Forth is not a suitable language to use with Windows, Linux or C libraries!

The four essential words needed are:

a) A word to define a z-string - currently:
: Z" (Comp: "ccc<quote>"--- ; Run: ---zaddr)
However, I note that the simple word " is still unused in standard Forth, so why don't we use that?

b) A word to type the z-string (used only for debugging) - currently:
: Z$. (zaddr---)
But see section 18 below.

c) A word for concatenating zstrings - currently:
: Z+ (zaddr1,zaddr2---zaddr3)
Note that this word MUST be thread safe, and must not involve any garbage collection. ***Also see section 18 below.***

d) A word to format a number as a zstring - currently:
: ZFORMAT (n---zaddr)

There are 3446 instances of Z", 1482 instances of Z+ and 432 instances of ZFORMAT in our main application.

Also frequently used, are:
: ZDIGITS (n1---n2) \ If n1 is zero, return zero. Otherwise assume n1 is the address of a zero terminated string, and convert the string to a number.
: Z= (z$1,z$2--- f) \ True if two zstrings are the same
: Z<> (z$1,z$1---f) \ True if two zstrings are not equal
: ZCAT (z$1,z$2---z$1) \ Concatenates z$2 to the end of z$1
(The user responsible for the buffer z$1 in the above)
: ZINITIATOR (zaddr---zaddr') \ Find start of a zero initiated string
: ZTRAILING (z$---) \ A zstring is adjusted to exclude trailing spaces
: ZLIMIT (saddr,maxlen---) \ Adds a zterm to a string of specified maximum length
: ZMOVE (src,dst---) \ According to the documentation, all this does is show off the VFX optimiser!!

In addition, a subset of the file access words are, as needed, converted to use zero terminated names.

12. UTF8

Forth needs to realise that the world has moved on from ASCII. The use of UTF8 is almost universal. Fortunately, UTF8 strings are single zero terminated, so all the zstring words above still work.

I am not quite sure what the XCHAR wordset in the Forth standard is for. All our applications are dynamically multilingual - yet we have never needed any of the XCHAR words.

One thing definitely needs fixing - C@. What this ought to do is fetch the UTF8 character at the address. If byte acting words are really needed, they should be B@ and B!.

13. Threads

These days, only the most trivial applications run in a single thread - yet Forth has no standard wordset for handling threads. Anyone approaching Forth for the first time would be under the impression that Forth does not do threads!

VFX does of course have quite good thread support, but there are three important improvements that are needed.

a) "User" variables (effectively thread-local variables) need to be converted to thread-local values.

b) A method of initialising the thread-local values is needed.

c) A better method of dealing with thread-local memory is needed.

There are 40 threads in our main application, though typically only half a dozen are running at the "same time".

14. Execution chains

This has been a feature of MPE Forths for many years, but it is only recently that we have discovered how extremely useful it is. This is another example of a concept that is easy in Forth, but quite hard if not impossible in most other languages.

15. Libraries and externs

Yet again, we have a situation that the Forth standard does not mention something that is a necessity for any serious application!

16. Databases

And again, how many real life applications need to access databases? The Forth Query Language (FQL) has been around for years and is rock solid. Dare I have the impertinence to suggest it should go into the standard?

17. Locals

A frequent criticism of Forth is that stack manipulation makes it hard to read. Stack manipulation should be de-emphasised in favour of locals.

The Forth standard needs to be completely rewritten to use:
{ <ins> | <locals> -- <outs> }
where standard ins and locals behave like VALUE (with all modifiers permitted), and other data types (e.g. float, string) are available. Locals should be automatically initialised.

17. Doubles

Think of one of the most popular and inexpensive tiny computers - the Raspberry Pi. It has a 64 bit CPU. Do we really need double length integers any more?

Let us try and think of some very large numbers, for example the US national debt. At the instant of writing, this stood at $37,289,586,478,935. The largest unsigned number representable in 64 bits is 18,446,744,073,709,551,615. We could express the US national debt in cents and still have plenty of headroom!

The reason for raising this issue is because when we demonstrate interactive Forth to a newcomer, they quickly understand it.

```
2 2 + . 4 ok        Nice!
```

But shortly afterwards, they will try:

```
1.2 3 + . 3 ok-1    WHAT???
```

That takes a lot of explaining, and it is completely unnecessary.

18. Numbers, and things

This brings me to possibly the most radical proposal, which is to address the issue of why 1.2 3 + does not work, when it so easily could work.

The first and easiest fix is that when we are free from double numbers, the floating point recogniser could accept 1.2.

The next thing to think about is data types. From the birth of Forth, there was an assumption that *everything* was integer. There was no need to consider data types, because there was only one.

We were told that floating point was slow, used a lot of expensive memory, and was unnecessary.

This has not been true for decades. Floating point is just as quick as integer, uses the same amount of memory, and is essential. So a floating point wordset was tagged onto Forth, and you had to remember to put an F in front of everything.

Furthermore, type conversion is manual, and you have to remember where you are on two different stacks.

There are 544 instances of S>F and 144 instances of F>S in our main application. There are no instances of F>D or D>F. This leads us to consider that maybe in the future floats should be the default, and that to specify an integer you need to put a prefix, just like you do to specify a hex number.

A more interesting idea might be to merge the data stack with the floating point stack, and instead add a type stack. The basic addition word, and many others, could then become type smart. Now we are away!

```
1.2 3 + . 4.2 ok
" abc" " 123" + . abc123 ok
3 1 3 WITHIN . True ok \ YEA!!
" 欧洲会议" 2 + . 会议  ok  \ That is "European conference", by the way.
```

Against that, there is sure to be the argument of speed. However:

a) Nowadays, most computing time is spent inside library calls, not in Forth itself.
b) If your program runs too slowly, spend 5 Euros on a faster PC - it's much cheaper than buying a programmer's time.
c) Constant arithmetic of mixed type as in the examples above would in any case be resolved by the optimiser at compile time, not at run time.
d) If you really, really needed to speed up some critical routine, the explicit arithmetic and type conversion words would still be available.

19. Conclusion

I hope that these notes and observations will lead to some radical changes to the Forth standard very quickly, so as to restore it to being a language of choice for discerning programmers.

EuroForth 2025

Short paper

Improvements to enumeration

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. Introduction

At Euroforth 2023 I proposed for standardisation a new enumeration wordset:

```
ENUM<< <enumname>
  [<Forth expression>] <membername> [\ <comment>]
  ...
>>
```

For example, one could do:

```
ENUM<< TESTENUM         \ Name of the enumeration
            AZERO       \ By default, the enumeration starts at zero
            AONE        \ Standard Forth comments are allowed
  1 2 +     ATHREE      \ Any Forth expression can be used to set the enumeration
            AFOUR       \ The enumeration increments
>>                      \ Enumeration terminator

TESTENUM SHOWCHAIN
AFOUR
ATHREE
AONE
AZERO  ok
```

This was well received by my colleagues. But it wasn't long before requests for extra features came along.

2. Add translated descriptions to the enumeration

A translated description of an enumerated value is a frequent requirement, and this was normally done in a separate word e.g. for the above example, it might have been:

```
: TESTENUMDESCR ( enval---z$ ) \ Returns translated phrase describing enval
  CASE
    AZERO   OF P" Zero"       ENDOF
    AONE    OF P" One"        ENDOF
    ATHREE  OF P" Three"      ENDOF
    AFOUR   OF P" Four"       ENDOF
    ^NULL
  ENDCASE
;
```

Clearly it would have been a lot easier to define the description phrase from within the enumeration, rather than in a separate word. So now we have:

ENUM<< <enumname>
   [<Description>] [<Forth expression>] <membername> [\ <comment>]
   ...
>>

But now there are two optional items before the member name. How can we possibly tell them apart, give that Forth has no data types? In particular, <Description> cannot consist of a phrase number, because
a) The phrase number could theoretically be quite a small number, well within the likely range of enumeration numbers.
b) During the build process, some enumerations are needed before we build the database access wordset, so that translatable phrases are not available at the point of definition of the enumeration.

This was a challenge, until we realised that when a zero terminated string is defined using Z" , you always get an address that is nowhere near HERE, which is where it always used to be. Strings are in fact always presented on a recently invented space called SYSPAD.

Since SYSPADSTART is typically a very large number e.g.

```
SYSPADSTART . 140734512400720  ok
```

this now gives us a way of distinguishing the two data types int and string, in all cases of int that are likely to be enumerated.

We could now get as far as:

```
ENUM<< TESTENUM               \ Name of the enumeration
                  AZERO       \ By default, the enumeration starts at zero
                  AONE        \ Standard Forth comments are allowed
  1 2 +           ATHREE      \ Any Forth expression which has a stack effect...
                              \ ...( ---n ) can be used to set the enumeration
                  AFOUR       \ The enumeration increments
  Z" Customer" 11 AN11        \ Description and enumval
  Z" Category"    A12         \ Just a description
                  A13         \ Neither
>>                            \ Enumeration terminator
```

Our enumeration recogniser now looks like this:

```
: ENUMINTERPACTION ( ??,caddr,u--- ) \ Interpreter action for enum recogniser
  ^NULL -> ENUMZ$                               \ Assume no description
  DEPTH 2 - 0 ?DO                               \ Deal with any preceding values
    ROT DUP SYSPADSTART DUP /SYSPAD + WITHIN IF \ It is an address within the
                                                \ strings buffer area
      -> ENUMZ$                                 \ Use it as a description
    ELSE                                        \ Probably not a string
      -> ENUMVAL                                \ Use it as a new enum value
    THEN
  LOOP
  ($CREATE)                                     \ Create the enumerated name
  ENUMVAL ,                                     \ Set the constant value
  0 ,                                           \ Reserve space for phrase number
  ENUMZ$ ZCOUNT Z$,                             \ Compile description string
  INC ENUMVAL                                   \ Next enumeration number
  LATEST-XT ENUMLIST ATEXECCHAIN                \ Add to list
  ['] ENUMVALCOMP, SET-COMPILER                 \ When an enumerated constant is
                                                \ being compiled
  INTERP> ENUMVALINTERP                         \ When an enumerated constant is
                                                \ being interpreted
;
```

We can still only save the original description though, not the translatable phrase number, which is not yet available. We've just left a space for it.

You will see that we create a list of all members of each enumeration, and there is a similar list of all the enumerations too.

It was not clear at the time precisely how these lists could be used - but now they proved to be really useful.

Right at the end of the build process, by which time all enumerations have been defined, and the database is up and running, we can execute a word that loops through all the enumerations and their members. It extracts the original description and matches it to a phrase number, creating new translatable phrases as necessary. It then pops the phrase number into the previously reserved space.

We have previously defined two new **modifiers** (I do wish they were not called operators in VFX), which enable us to easily access the original text and the phrase number of any enumerated member.

```
OPERATOR: ENUMPHRASE    \ Returns the phrase number of an enumerator
  OP# ENUMPHRASE CONSTANT OPENUMPHRASE
OPERATOR: ENUMDESCR     \ Returns the address of the description
  OP# ENUMDESCR  CONSTANT OPENUMDESCR
```

3. Making enumerated values available in external database queries

The second request from my colleagues was that enumerated values should be available automatically in the database. Generally, our main application, in Forth, is supported by several "dashboard" apps. The Forth program controls the system and places reportable information into the database. The dashboards, which require no programming, just configuration, display live data. Part of the configuration is the provision of an SQL statement that the dashboard can use to extract the data it needs. Previously, the SQL statements were littered with "magic numbers" representing our enumerated values. Every time a change was made to an ENUM<< in the Forth code, the dashboard configurations had to be checked in case any magic numbers had changed.

The solution was to create, automatically, a "loadable function" in the database, for each enumeration member. Then, a function can be used instead of a magic number inside an SQL query, and the results always match. For example

```
REPEV_CHCUS . 92  ok
SQL| SELECT REPEV_CHCUS() |SQL>>
+---------------+
| REPEV_CHCUS() |
+---------------+
| 92            |
+---------------+ ok
```

We can now take a look at a simplified version of the word which does all this, right at the end of the build.

```
: SETENUMPHRASES \ Place phrases for enumerations and create DB function
  { | penumname[ 255 ] pelementname[ 255 ] pelementnum plementdescr[ 255 ]
      pphrase -- }
  ENUMSLIST @ BEGIN                                     \ Anchor of enumerations
  DUP WHILE                                             \ Another enumeration
    DUP CELL+ @                                         \ Get xt of enumeration
    DUP IP>NFA 1+ penumname[ ZMOVE                      \ Get name
    EXECUTE @ BEGIN                                     \ Get anchor of elements
    DUP WHILE                                           \ Another element
      DUP CELL+ @                                       \ Get xt of element
      DUP IP>NFA 1+ pelementname[ ZMOVE                 \ Get name of element
      >BODY                                             \ To element data
      DUP @ -> pelementnum                              \ Get element number
      DUP 2 CELLS+ plementdescr[ ZMOVE                  \ Get description
      SQL| DROP FUNCTION IF EXISTS                      \ Discard old function
           | pelementname[ >SQL |
      |SQL
      plementdescr[ C@ IF                               \ Description is defined
        SQL| CREATE FUNCTION                            \ Create new function
             | pelementname[ >SQL | ()
             RETURNS INT
             DETERMINISTIC                              \ If replication used
             RETURN | pelementnum FQL-N+ |
        |SQL
        plementdescr[ FINDPHRASE -> pphrase             \ Get phrase number
        pphrase SWAP CELL+ !                            \ Set in element data
      ELSE                                              \ No element description
        DROP                                            \ Address of data
      THEN
      @                                                 \ Get next element
    REPEAT DROP                                         \ Discard element chain
    @                                                   \ Get next enumeration
  REPEAT DROP                                           \ Discard chain
;
```

4. Conclusion

A lot of this would have been easier if data types were more easily available - see my next paper "Forth 2025".

Code-Copying Compilation in Production An Experience Report

M. Anton Ertl* Bernd Paysan
TU Wien net2o

Abstract

A code-copying compiler implements a programming language by concatenating code snippets produced by a different compiler. This technique has been used in Gforth since 2003, with code snippets generated by GCC. We have solved various challenges: in particular, which code snippets can be copied and what to do about the others; and challenges posed by changes in compilers. The performance of Gforth is similar to that of SwiftForth, a commercial system with a conventional compiler; the implementation effort is comparable to 1–2 targets for SwiftForth.

1 Introduction

Code copying is a programming language implementation technique where the compiler of the implemented languate A concatenates code snippets coming out of the compiler for language B. While there have been a number of research papers about this topic (see Section 8), we know of only one production language implementation that has used this approach for a long time: Gforth.

The present work is an experience report about the use of code copying in Gforth: How does it compare to a conventional compiler (Section 2)? Section 3 explains the concepts of code copying, while Section 4 discusses various implementation aspects. We also discuss the problems from changes in compilers (Section 5) and operating systems (Section 6) and how we overcame them.

In addition to this experience report, this paper also discusses alternative approaches (Section 7) and related work (Section 8).

The present work also appears in the KPS 2025 proceedings, with the same content and different formatting.

1.1 Is Gforth a production system?

Gforth is free software that has been developed since 1992 and first released in 1996. As it is free software, everybody can use it without contacting us, and few people do, so we do not know that much about who uses it for what purpose. However, we know that it has been used by IBM and Apple in their work on Open Firmware, and Forth, Inc. (who develop SwiftForth, but also give Forth courses) have given courses using Gforth, also in the Open Firmware context. So: Yes, Gforth is a production system.

2 Why not just write a conventional compiler?

One reason why people may have avoided going for a code-copying compiler is the assumption that writing a conventional compiler will produce better code, or require less effort. By “conventional” we mean that there is a large amount of hand-written architecture-specific code for each target architecture in the compiler. So before we go into details about code copying, we will address this concern.

2.1 Performance

Figure 1 shows the performance of the `gforth-fast` engine of Gforth[1] with various optimizations, of two commercial conventional Forth compilers (SwiftForth and VFX Forth), and, for of GCC-12.2 `gcc -O0`, `-O1`, and `-O3`. All Forth systems use load-and-go compilers (compile time is included in the results), while GCC uses ahead-of-time compilation (only the run-time is shown in the results).

Not all benchmarks are available in C, and not all benchmarks run on all Forth systems, and the missing cases are reflected by missing bars.

The data shown is the median of 30 runs for each benchmark/system combination on a Core i5-1135G7 (Tiger Lake); each bar represents the number of cycles of Gforth with only code copying divided by the number of cycles of the system represented by the bar, i.e., the speedup of that system over Gforth with only code copying. The Gforth version used is `0.7.9_20250817`, commit `4224ab5fafea970dade64b04493ef690da8b3c32`

*`anton@mips.complang.tuwien.ac.at`

[1] Gforth also has an engine `gforth` intended for debugging. All references to Gforth performance refer to `gforth-fast`.

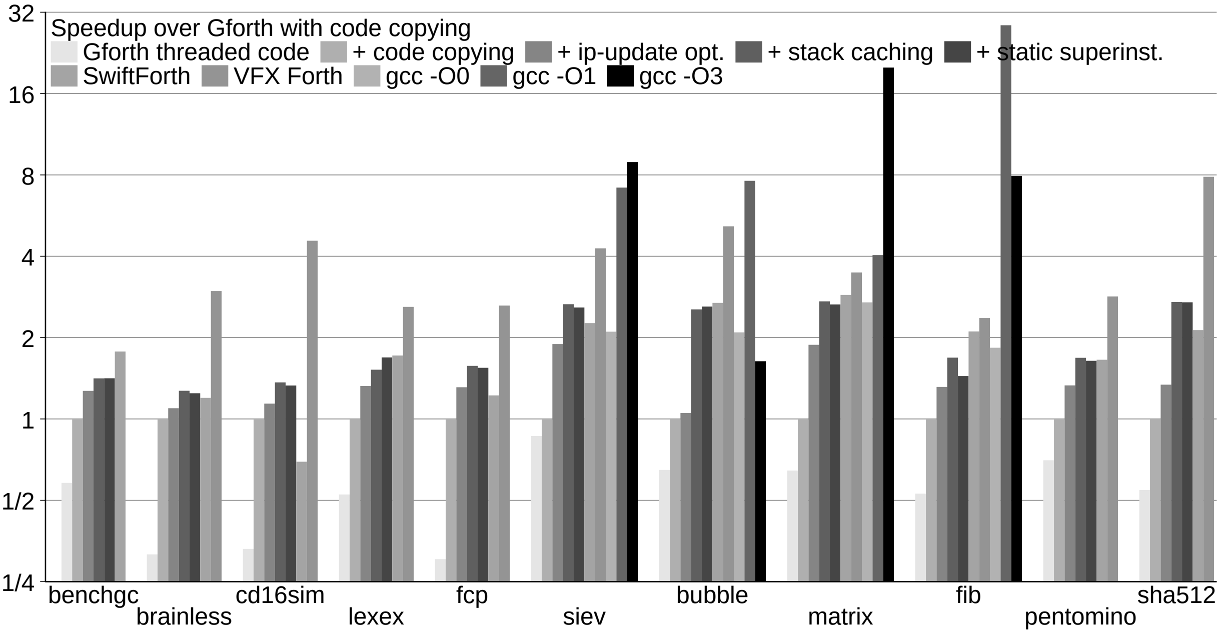


Figure 1: Speedup factor of various systems over Gforth with code copying, on a Core i5-1135G7 (Tiger Lake)

compiled with gcc-11.4. The benchmarks are from the Forth appbench suite (benchgc–fcp), Gforth's small (and mostly loop-dominated) benchmarks (siev–fib), and two additional ones.

As can be seen, the performance of Gforth with all optimizations is similar to that of SwiftForth, which uses a conventional compiler, and typically around half of the performance of VFX Forth, which also uses a conventional compiler.

Before comparing Gforth with the others, let's first take a look at the variants of Gforth, starting with the one with the best performance/effort:

Threaded code This is a fast interpretation technique for virtual-machine (VM) code, without any machine-code generation (see Section 3.1).

Code copying This method concatenates code snippets from the threaded code engine (see Section 3). It requires an estimated 500 lines of code in the Gforth source code. With this method Gforth still accesses literal data and performs control flow by accessing the VM code; it therefore also maintains a VM instruction pointer (IP), and updates it once for every VM instruction.

IP-update optimization This optimizaton reduces these IP updates. It was added by inserting 864 lines and deleting 316 lines in the Gforth source code [EP24].

Stack caching (actually static multi-state stack caching) eliminates many memory accesses to stack items and stack-pointer updates [EG04a, EG05]. The way this optimization as implemented in Gforth requires code copying to work.

Static superinstructions replace a sequence of Forth words with an optimized sequence [EGKP02]. Many of the benefits that static superinstructions have originally provided are now provided by code copying, the IP-update optimization and static stack caching; there are still cases where static superinstructions result in shorter code, but this has not led to consistent speedups in these measurements.

The code implementing stack caching and static superinstructions is quite interweaved with the rest of the code, so it is hard to give precise numbers for their size, but we estimate [Ert24] that all four optimizations combined require an estimated total of 5000 lines of code.

SwiftForth's compiler can be seen as a copy-and-patch compiler, but with the code snippets written by hand in assembly language and better resulting code than when patching using object file linkage imformation (see Section 7.3). SwiftForth does not have a VM interpreter substrate, and therefore does not have IP updates, so it gains the benefits of the IP update optimization without having to do anything. It deals with literal values and control flow by patching the code. SwiftForth does not perform multi-state stack caching, but it makes extensive use of static superinstructions (346 rules in 1819 lines). Overall each of the IA-32 and

AMD64 targets of SwiftForth has about 7000 lines of architecture-specific code [Ert24].

Gforth with all optimizations is competetive in speed with SwiftForth, so apparently Gforth's stack caching provides enough speedup to compensate the costs that Gforth incurs for literals and control flow.

VFX Forth performs register allocation of data-stack items within a basic block, and inlines aggressively; inlining is very helpful for idiomatic Forth code, where calls and returns are the most frequent basic block boundaries. Therefore inlining also enhances the effectiveness of VFX's register allocator. The speed advantage of VFX over Gforth and SwiftForth is a result of these optimizations. In particular, for the cd16sim benchmark there is one call site that calls an empty definition and that is responsible for 2/3 of Gforth's run-time on this benchmark, while VFX inlines it away. We have no source code for VFX and therefore cannot report numbers about the size of its compiler. When asked about the effort to port VFX to ARM A64 (a currently ongoing project), Stephen Pelc gave the qualitative statement "far too much".

VFX is faster than Gforth by typically around a factor of 2. However, it is possible to perform inlining in Gforth, too, with direct performance benefits as well as indirect benefits from better stack caching. It will be interesting to see how far Gforth (and code copying) can close the gap.

Gforth's performance with all optimizations is roughly comparable to that of `gcc -O0` on those benchmarks that are also available in C. `gcc -O1` and `gcc -O3` often produce significantly faster code; sometimes they don't, but the reasons for that are beyond the scope of this paper.

2.2 Portability

A major reason to avoid implementing a conventional compiler is portability/retargetability.

Gforth has supported as many architectures as we could get our hands on, as long as gcc and something Unix-like (e.g., Cygwin for Windows) is available on the architecture. Gforth has supported the following architectures with a code copying compiler: Alpha, ARM A32/T32, ARM A64, HPPA, IA-32, IA-64, Loongarch, SPARC, PowerPC, PowerPC64 (but we no longer can check for all architectures that they still work). Gforth supports all architectures it does not know about by falling back to threaded code, which is slower, but still works.

In particular, when IA-64 (launched 2001) and AMD64 (launched 2003) became available to us in 2003, Gforth worked out of the box on these architectures[2] using the unknown-architecture support, likewise for ARM A64 in 2014 and RISC-V in 2017. A few small changes enabled code copying[3], and a one-line change for configuring the number of registers for stack caching.

The benefit of code copying is that it reuses the retargeting efforts of the compiler it is based on (GCC or Clang in case of Gforth).

By contrast, SwiftForth has supported only IA-32 until the 2020s, when they started working on an AMD64 port (released on 2025-10-22). VFX has supported IA-32 initially, later ARM A32, and, also starting in the 2020s, AMD64. Both systems have interactive cross-compilers for a number of embedded targets.

The low number of desktop ports and the late support for AMD64 may be due to lack of commercial interest, but we think that the larger effort required to retarget and maintain the compiler for another architecture has something to do with it. iForth, another conventional Forth compiler, got an AMD64 port in 2009, but the IA-32 port was subsequently dropped (last release with IA-32 support in 2011).

2.3 Incremental development

Another benefit of code copying over writing a conventional compiler is that it can be done step-by-step: First add code copying, then add one optimization (e.g., IP-update optimization), then the next, etc., always with the fallback options of disabling the optimization or completely falling back on threaded-code interpretation.

By contrast, when coming from an interpreter, the conventional model requires a big-bang approach where a complete code generator for one target has to be developed without reusing much from an existing interpreter; and as long as you do not develop code generators for all targets, you still need to maintain the interpreter, as well as all the compiler targets. The latter will hopefully be helped by designing the compiler for retargetability, but that increases the complexity of the compiler framework.

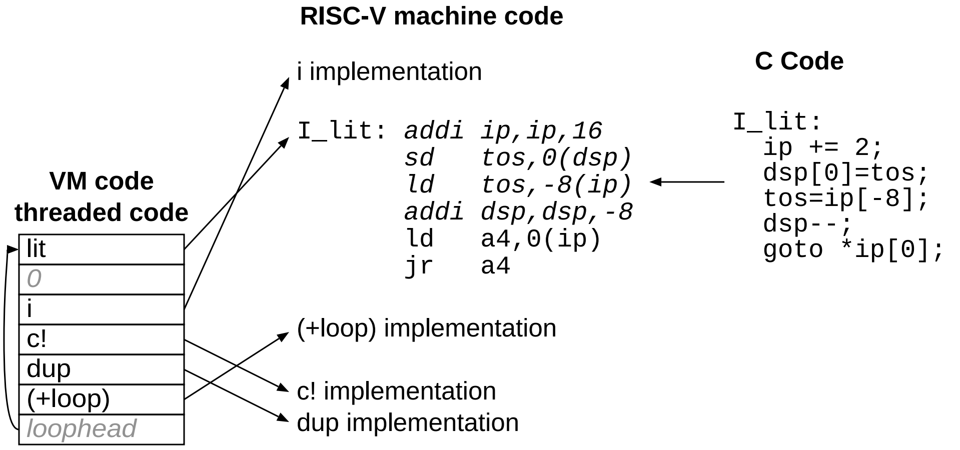
3 What is code copying compilation?

3.1 Threaded Code

The basis for Gforth's code-copying implementation is a threaded-code interpreter [Bel73] for Gforth's virtual machine (VM).

[2] We added 64-bit support in 1996 while doing the Alpha port.

[3] For RISC-V, this was our first encounter with gcc-7 and its more aggressive code duplication (Section 5.4); we needed a little longer to find a workaround for that, but that's not specific to the architecture.



```
I_lit: addi ip,ip,16
       sd   tos,0(dsp)
       ld   tos,-8(ip)
       addi dsp,dsp,-8
       ld   a4,0(ip)
       jr   a4
```

```
I_lit:
  ip += 2;
  dsp[0]=tos;
  tos=ip[-8];
  dsp--;
  goto *ip[0];
```

Figure 2: Threaded-code representation of VM code. Each box is a machine word. *Slanted light blue* indicates an immediate operand of the preceding VM instruction.

As a running example, we look at the VM code in Fig. 2. The first VM instruction in the example is `lit`, which has an immediate operand *0*. This VM instruction pushes its immediate operand on the data stack. It is represented by the address of the machine code that implements it; in direct-threaded code, every VM instruction is represented by the address of the machine code that implements it. In the case of `lit`, the implementation for RISC-V (RV64G) is:

```
                    # //C code
addi ip,ip,16       # ip += 2;
sd   tos,0(dsp)     # dsp[0] = tos;
ld   tos,-8(ip)     # tos = ip[-1];
addi dsp,dsp,-8     # dsp--;
ld   ca,0(ip)       # ca = ip[0];
jr   ca             # goto *ca;
```

This code uses register names that reflect their roles: `ip` is the VM instruction pointer; `tos` is the top of the data stack; `dsp` is the data stack pointer; `ca` is the code address (of the next VM instruction).

The *slanted blue* instructions are the payload which perform the actual work of the VM instruction as far as code copying is concerned. Other optimizations reduce that part further; e.g. the first instruction updates IP, and the IP-update optimization often optimizes it away.

The third instruction loads the immediate operand (0) from the VM code by accessing it through IP. This access of immediate operands and control-flow operations through IP is still in Gforth with all optimizations applied, and is the difference between an interpreter-based code-copying system and a copy-and-patch system (Section 7.3).

The bottom two (black) instructions perform the dispatch to the next VM instruction. The first instruction loads the machine code address of the next VM instruction, and the second instruction jumps to it.

This assembly-language code can be generated from the C code shown in the comments of the assembly language. It uses the GNU C extension “Labels as Values”,[4] which allows jumping to the address in `ca` with `goto *ca`[5]; this extension is also supported by Clang, tcc, and icc.

The other VM instruction implementations have the same pattern of *payload*, and dispatch. The last VM instruction in our example, `(+loop)` is notable: it is a VM-level conditional branch that branches back to *loophead* (given as immediate operand) or falls through to the next instruction. It is implemented with the following code

```
 addi ip,ip,16            # ip += 2;
 ...compute condition...
 blt  a5,zero,fallthrough # if (taken) {
 ld   ip, -8(ip)          #   ip = ip[-1];
 ld   ca, 0(ip)           #   ca = ip[0];
 jr   ca                  #   goto *ca;
fallthrough:              # }
 ld   ca,0(ip)            # ca = ip[0];
 jr   ca                  # goto *ca;
```

If the conditional branch is taken, the new IP is loaded from the immediate operand and a dispatch is performed. It is better to have separate dispatches for the taken and the fallthrough cases for branch prediction[6] and because it allows to leave away the fallthrough dispatch in code-copying.

[4] https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html

[5] The GCC maintainers call this a computed goto, although it is more like a Fortran assigned goto.

[6] Even with history-based indirect-branch prediction, branch predictors have an easier time if there are fewer targets for each indirect branch

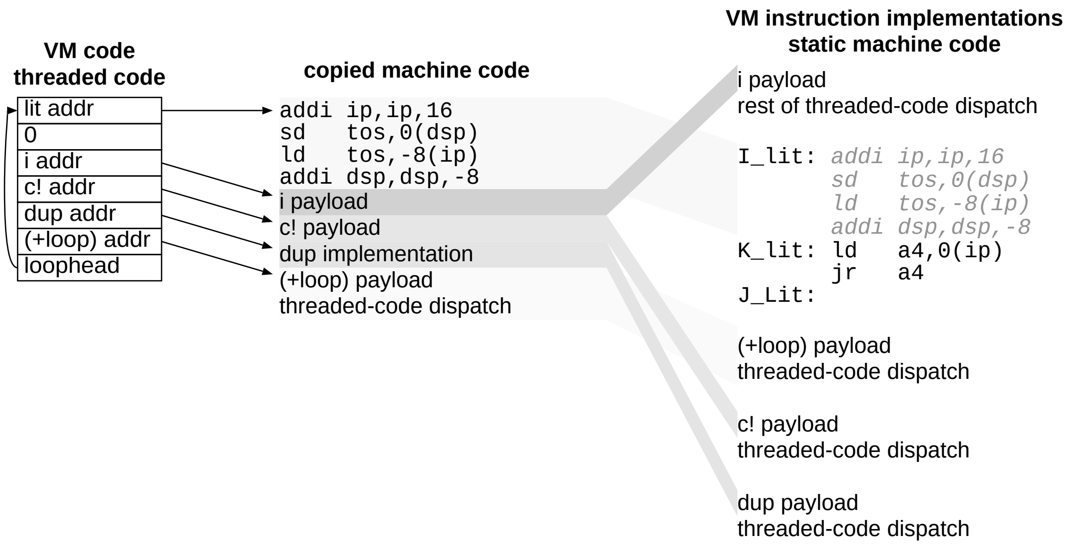


Figure 3: Code copying.

3.2 Code copying

Most VM instructions do not perform VM-level control flow, but just continue with the next VM instruction. Code copying copies and concatenates the machine code implementing the VM instructions, but in most cases without the dispatch code at the end. Only taken branches (i.e. VM instructions that change IP to point to some other VM instruction than the next one) need to perform a dispatch.

Figure 3 shows this for our running example. The VM code is conceptually the same as before, but for each VM instruction the machine word now points to the copied machine code instead of the original.

In particular, the copied code still has the IP, which points to the threaded (VM) code, and it accesses the immediate operands *0* and *loophead* through it. The threaded code is also used on control flow: the VM-level conditional branch `(+loop)` is taken, loads the target threaded-code address *loophead* into IP, and then performs a threaded-code dispatch, which loads the code address at loophead, which points to the start of the concatenated code. All control flow in Gforth is performed with threaded-code dispatches in this way.

The threaded-code slots for instructions other than `lit` in this example are not accessed during execution. Gforth keeps them around to simplify the implementation.

At the end of the shown sequence the threaded-code dispatch is copied. While this is necessary for unconditional branches, it is not generally necessary for conditional branches such as `(+loop)` (as discussed above). However, the following VM instruction may make it necessary to perform a dispatch after the `(+loop)`.

Code copying has also been called the `memcpy()` method [RS96], selective inlining [PR98] and (especially in Gforth) dynamic superinstructions [EG03a].

3.3 Benefits over threaded code

The obvious benefit of code copying is that it eliminates most threaded-code dispatches and results in straight-line execution of VM-level straight-line code, avoiding the limit of typically one taken branch per cycle. Another benefit is that the indirect branches in most of the remaining dispatches have only one target, vastly improving branch prediction accuracy in CPUs without sophisticated indirect-branch predictors, and still making life easier (and faster) for hardware with such branch predictors.

Another benefit is that code copying enables additional optimizations that require code snippets that are not represented as VM instructions (and where introducing additional VM instructions with threaded-code dispatch would make the optimization unprofitable).

E.g., the IP update optimization [EP24] leaves the IP update in front of most VM instruction implementations away and replaces it with an IP update by a larger amount for VM instructions that actually use the IP.

As another example, stack caching as implemented in Gforth inserts transitions between stack-cache states where necessary. These transitions do not have a VM instruction slot and therefore can only be inserted when code-copying is enabled. Gforth's stack-caching implementation relies on being able to insert the transitions, so stack caching is disabled when code copying is disabled [EG04a].

3.4 When is code copying appropriate?

The shorter the VM instruction implementations are, the larger the benefit of code copying over threaded code, because the overhead of threaded-code dispatch is relatively larger then.

Conversely, with long VM instruction implementations as in Tcl, whose VM instructions "can average hundreds of [machine] instructions" [VA04] the benefit is small, and often does not amortize the cost of copying the code or of increased I-cache misses [VA04].

Another aspect is that a compiler (to VM code) that uses more VM instructions, with each doing less, has more opportunities to optimize the VM code. This has been done for CPython recently[7]. With expensive VM instruction dispatch, splitting an existing VM instruction into several simpler ones increases the cost, and the opimization must be very good and must be applicable often to amortize this cost. With code-copying, the dispatch cost approaches 0, and such transformations become less of a gamble.

4 Implementation of code copying

4.1 Code organization

Gforth has a big function `engine()` that contains all the code snippets (implementations of all VM instructions, and additional snippets used by optimizations), and little else.

Every code snippet has a label in front of it and behind it:

```
L_before:
  code snippet in C;
L_after:
  threaded-code dispatch;
```

You can see that more concretely in Fig. 3.

The label before it obviously points to the start of the code snippet.

Getting the right label for the end of the code snippet was initially straightforward (up to gcc-3.1), but later required extra work. If the source code falls through to the label (i.e., it does not end in an unconditional branch), like for the payload of most VM instructions in Gforth, with some extra help (see Section 5.4), the following label points right behind the code snippet, but if the code snippet cannot reach the label (e.g., because it ends in an unconditional branch, e.g, in a threaded code dispatch), gcc-3.2 and following have reordered code. We solved this problem by taking the values of all the labels, sorting them, and searching for the first label behind the label at the start of the snippet. This might include some unrelated code in cases where the code snippet does not fall through to the label, but in that case this is not a problem for correctness (but possibly for relocatability, see Section 4.5).

The function `engine()` has two code paths: the first just returns a table containing all the labels, for use in threaded-code generation and code-copying; the second starts the execution of the code by performing a threaded-code dispatch.

If code copying is disabled,[8] the threaded code address for each VM instruction just points to the implementation of that instruction inside `engine()`, and every threaded-code dispatch jumps around within this function.

With code copying, the first threaded code dispatch in `engine()` jumps to the copy of the VM instruction implementation and continues running there, with control-flow changes by performing a threaded-code dispatch.

4.2 Why does it work?

Why can we concatenate the code snippets produced in the way described above, and get code that works?

In particular, won't the register allocator have different register allocations for the different code snippets? Actually, at the start and, for fallthrough snippets, the end of the snippet, the register allocation has to be the same as at the start of every other snippet, because the compiler has to consider the possibility that every `goto *` jumps to every label whose address is taken. And the addresses of all labels before and after all code snippets are taken (to determine the code snippet address and length).

The code snippets that do not fall through end in a `goto *` in Gforth. And the register allocation at the `goto *` has to be compatible with that of all the labels whose address is taken, or it would not work even in ordinary use.

More precisely, `engine()` is compiled separately from the code dealing with the threaded code, so the C compiler has to assume that every `goto *` in `engine()` can jump to any label whose address is taken.

Therefore, at a `goto *` all variables are alive (i.e., read before being overwritten) that are alive at any label whose address is taken, and each variable has to be in the same location at all those labels and all the instances of `goto *`. The code snippets that fall through to their second label are followed by a threaded-code dispatch:

[7] `https://github.com/faster-cpython/`

[8] Gforth option `--no-dynamic`.

```
    ca = ip[0];
    goto *ca;
```

so at the label between the code snippet and the dispatch, all the same variables are alive as at the goto *, except possibly ca, but that is not alive before the threaded-code dispatch, either. These variables also all have to reside at the same locations, because the goto * could jump to them.

4.3 Fallback

There are cases where certain code snippets cannot be copied (usually because they are not relocatable, see Section 4.5). How does Gforth deal with that?

Gforth falls back to plain threaded code in these cases: Append a threaded-code dispatch to the previous copied code snippet (unless the code snippet already ends with a threaded-code dispatch), and let the machine word representing the current VM instruction point to the original implementation of the VM instruction (inside engine()) rather than a copy). At run-time, the code performs the threaded-code dispatch, which then jumps to the original; that ends in another threaded-code dispatch, which may jump to code coming out of code-copying, or to another original implementation.

If other optimizations are active, the preparation for the fallback may require appending additional code. E.g., the IP needs to be up-to-date before the threaded-code dispatch, so in the presence of IP-update optimization, an IP update may be inserted before the threaded-code dispatch. Also, in Gforth the plain threaded code always expects the stack in the canonical state, so in the presence of stack caching, a transition from the current stack state to the canonical stack state may need to be inserted before the threaded-code dispatch.

Gforth may also find that it cannot copy the threaded-code dispatch. In that case it disables code copying completely and falls back to threaded code not just for individual VM instructions, but for all of them.

The option to fall back to threaded code has helped in various cases where things did not work according to our expectations (e.g., see Section 5.4). It means we always have a way to make Gforth work, albeit not as fast as we would like.

4.4 Instruction sets

Code copying is based on the assumption that the code snippets are independent and concatenable. At the instruction-set level this is satisfied if individual instructions are independent and concatenable. Some instruction sets have restrictions between groups of instructions. In this case a code snippet must not contain a partial group, i.e., there must not be a label within a group.

There are a few cases of such instruction-set restrictions:

Branch delay slots This is a misfeature of some early RISC architectures, in particular, HPPA, MIPS and SPARC: The branch instruction performs the instruction behind it before continuing at the target. This does not work with code copying if the compiler puts a label between the branch and the instruction behind it. However, the compilers we have used (most recently gcc-14.2) do not do that.

Load delay slots This is a restriction of the MIPS I instruction set (eliminated in MIPS II). The instruction behind a load instruction is not allowed to read the register written by the load instruction. MIPS I also has some placement restrictions on reading and writing the hi and lo registers. Having labels right after the load or in the shadow of hi/lo reads can result in violating these restrictions in code copying. We have not tested if compilers actually place labels in a way that would lead to such violations. Instead, these concerns along with the relocatability problems (Section 4.5) and the lack of relevance of MIPS in Unix systems around 2003 were the reasons why we just configured Gforth to fall back to threaded code on MIPS (including the 64-bit MIPS port).

Instruction groups This is an IA-64 (aka Itanium processor family) property. Instructions within a group have restrictions on register usage that are intended to ensure that the instructions can be performed in one cycle without register renaming.[9] If a compiler put a label inside a group, code copying could violate these restrictions. Apparently the compilers we used (gcc-3.3, gcc-4.1.3, gcc-4.3.2) put stops (group boundaries) at labels, because in our testing IA-64 has always worked fine. If they did not, an easy fix would be to insert the stops using asm statements or at the assembly-language stage.

Based on the experiences with branch delay slots and instruction groups, it seems that gcc developers also avoid splitting groups of instructions with interdependencies by inserting a label inside these groups, but if these instruction sets still were important targets, that might change.

[9] Groups are often confused with bundles, which are IA-64's encoding of three instructions in 128 bits. By contrast, groups can be arbitrarily long, and can start and end somewhere in the middle of a bundle.

The problematic restrictions/features have not spread to newer architectures and all the architectures with these restrictions in general-purpose computers have been canceled in the meantime, while older or contemporary architectures without these restrictions thrive. So apparently the idea of independent, concatenable instructions has some merit, and we can expect that future instruction sets will also exhibit this property and thus support code copying.

4.5 Relocatability

A code snippet must be relocatable in order to be used in code copying, i.e., it must behave the same way in the original place and when copied.

Non-relocatable code

The main problems here are references to addresses: The code in the snippet must refer to addresses inside the snippet in a PC-relative way, and must *not* refer to addresses outside the snippet in a PC-relative way. Most architectures refer to other code addresses in a PC-relative way, so the most common reason for non-relocatability is when the VM instruction implementation performs a call to some function (e.g., for performing I/O).

Accesses to global constants or to global variables in a PC-relative way can also cause non-relocatability. Gforth avoids global variables for that reason and because of multi-threading; it stores some formerly global variables in a struct whose address is stored in a local variable inside `engine()`. However, computing the FP negation and the FP absolute value implicitly involve a constant that resides in memory on AMD64 (with SSE2 FP), making the implementations of these VM instructions (`fnegate` and `fabs`) non-relocatable on this architecture.

The pointer-to-struct approach could also be used for invoking functions without making the calling code non-relocatable, but for now we have not done that.

Note that asking the C compiler for position-independent code does not mean that individual code snippets are relocatable, even though the binary as a whole is, because position-independent code may refer to code or data outside the code snippet in a PC-relative way (and usually does), while a relocatable code snippet must not do this.

Determining relocatability

How do we find out if a code snippet is relocatable or not? The implementations of the VM instructions actually look as follows:

```
L_skip:
  asm("SKIP4");
  asm("SKIP4");
  asm("SKIP4");
  asm("SKIP4");
L_before:
  code snippet in C
L_after:
  asm("SKIP4");
  asm("SKIP4");
  asm("SKIP4");
  asm("SKIP4");
  threaded-code dispatch
```

We compile `engine()` with these pieces to assembly language. Then we assemble the result twice: Once with `SKIP4` defined as empty string, so the `SKIP4`s assemble to nothing, and the result is as discussed earlier; and once with `SKIP4` defined as `.skip 4`, and with `engine` defined as `engine2`, so as a result the object file contains a function `engine2()` that has 16 bytes of padding before and after each code snippet.[10] We link both object files into the final executable. The addresses of the `L_skip` labels are taken and passed outside `engine()`, so gcc cannot optimize the initial skip away as dead code, and also because that usually is the next label after a threaded-code dispatch.

We now have a function `engine()` without the skips before and after the code snippets, and a function `engine2()` that has 16-byte skips before and after each code snippet. We extract the labels from each of the functions, and then compare the code snippets: If a code snippet from `engine()` contains exactly the same bytes as the corresponding code snippet from `engine2()`, then the code snippet is relocatable, otherwise it is not.

How does this work? If code from inside the code snippet references a code or data address outside the code snippet through a PC-relative address, the offset of the relative address will be different between `engine()` and `engine2()`, because the target label will be farther away in `engine2()` thanks to the skips. If there is an absolute reference (e.g., MIPS `j` instruction) to inside the code snippet, it will be different between `engine()` and `engine2()`, because the respective targets are at different addresses.

Even if the code snippet ends in an unconditional branch and the C compiler puts some other code behind that unconditional branch,[11] this scheme works: If the two code snippets compare equal, the

[10] In earlier times we compiled twice rather than assembling twice, but compiling once is faster, and we do not need to worry if the two compilation runs introduce unintended differences in addition to the intended ones.

[11] We have not seen such an occurence yet.

code is relocatable. When used in a code-copying system, the code snippet may have some unused code behind the unconditional jump, but the generated code is still correct.

The reason for skipping 16 bytes is that this is a common code-alignment value, so the skips would not result in altered alignment (these days we ask the compiler to align to 1-byte boundaries, so skipping less might be sufficient). The reason for performing the 16-byte skip as 4 4-byte skips is that for some targets gcc counts the number of instructions in `asm` statements, assumes that each instruction takes at most 4 bytes, and generates code that relies on this assumption.

The absolute target addresses for the MIPS `j` and `jal` instructions have a catch: They work only for targets in the same 256MB segment of the address space. When we last looked, the functions `engine()` and `engine2()` were linked in the same 256MB segment as the functions called by some of the code snippets, and the code snippets would have been classified as relocatable. However, they were only relocatable within this 256MB segment. This is another reason why we disabled code copying for MIPS. An alternative would have been to allocate the memory for the copied code in the same 256MB segment as the original. Fortunately, among the architectures we have looked at, only MIPS has this property.

5 Compiler issues

In the previous section we have already mentioned a few caveats about how compilers have interfered with our initial assumptions about the generated code, and what we do about that. This section discusses additional issues.

We had quite a few problems with various gcc versions in the 2000s, and for some we found ways to deal with them, while some others were eventually fixed (after reappearing for several years). Also, the rethoric about undefined behaviour started at around that time and has spread and become more aggressive since then,[12], so at some point we expected to have to switch from using GNU C to assembly language as a more reliable foundation at some point [Ert14], essentially switching to a conventional compiler. But this has not happened (yet?), and actually, in the 2010s and 2020s only few new problems have appeared, and we found ways to deal with them. So GNU C seems to be a relatively stable foundation after all, once one has implemented various workarounds.

5.1 Code reordering

When we started, gcc arranged the basic blocks in source order. This changed with gcc-3.2. This has an effect on how we find the next label (Section 4.1). But we also saw cases where the compiler moved basic blocks from between `L_before` and `L_after` to outside these labels, which caused problems.

To avoid such problems, we tried to have only straight-line code in the VM instruction implementations. We extracted loops and most `if`-statements into functions that are compiled separately, and the VM instruction implementation only contains a call to this function. This costs a little performance (from the function call as well as turning the VM instruction implementation into non-relocatable code on most architectures), but fortunately the VM instruction affected by this are executed relatively rarely.

However, conditional VM branches are executed frequently, and in the ideal case they contain a conditional branch, in the following form (also seen for `(+loop)` in Section 3):

```
  ... skips ...
L_before:
  ... stack handling etc. ...
  if (VM_branch_taken)
    ip = ip[-1]; /*VM-branch target*/
    threaded-code dispatch;

L_after:
  ... skips ...
  threaded-code dispatch;
```

Ideally such VM-instruction implementations are compiled such that the basic blocks in the machine code are in the same order as in the source code, so that the code controled by the `if` is between `L_before` and `L_after`, and the second threaded-code dispatch can be left away by code-copying in the usual case. For now, gcc does it that way for our code. But if gcc ever started changing this, a possible way to steer it back on the right path may be to use `__builtin_expect(VM_branch_taken,1)` instead of just `VM_branch_taken`.

5.2 Code alignment

Compilers insert padding to align branch targets to instruction-fetch boundaries or cache-line boundaries. In particular, they do this for branch targets behind unconditional branches and loop heads.

When code copying, the padding inserted for the original code is often inappropriate for the target code. Therefore, we suppress this padding by compiling `engine()` with the options `-falign-labels=1 -falign-loops=1`

[12] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

`-falign-jumps=1`.

Instead, our code-copying implementation performs its own alignment (but on 2007-era processors where we measured the effects, the effects were in the noise).

5.3 Code deduplication

Starting with gcc-3.0, gcc started to compile all the `goto *` instances to an unconditional jump to one instance of an indirect branch. The reason for this probably was to reduce the control-flow edges in the data-flow analysis, for m `goto *` and n labels from nm to $n + m$.

In a number of gcc versions (up to the early gcc-4.x releases), gcc then did not eliminate the unconditional jump afterwards, with some versions eliminating them and some versions regressing, but eventually the gcc maintainers managed to make the unconditional-branch elimination stick, for our code.

So if that is a solved problem, why do we mention it here? We occasionally see this problem reappear in some form, so it's not completely gone.

E.g., when we managed to extend stack-caching support on AMD64 to three registers, we found that on AMD64 gcc compiled the `goto *` to an unconditional branch to common code that contains a lot of register shuffling (with no overall effect) and finally the indirect branch. Apparently the register shuffling made the common code so long that the branch-elimination heuristic decided not to eliminate the branch.

Fortunately, we found out that the register shuffling (and, consequently, the unconditional branch) go away with the compilation option `-fno-tree-vectorize`. Apparently without this option gcc tries to vectorize loads and stores of adjacent values, and is less precise in the data flow analysis for that than for individual values, leading to the register shuffling.

For the problems in the gcc-3.x and 4.x era, Gforth contains a workaround that has just one threaded-code dispatch and jumps there from all the VM instruction implementations. Gforth has labels before and after this dispatch, and because there is only one, gcc does not deduplicate it; this allows Gforth to use it as a code snippet that is appended whenever a threaded-code dispatch is needed.

In order to work with this workaround and still be relocatable, we implemented conditional VM branches to just set the IP on a taken branch, and then continue through `L_after` to the dispatch code. This results in worse code than we would have liked, but it was the best that was possible on these compiler versions. This approach remains an option when building Gforth,

5.4 Code duplication

On our first encounter with gcc-7, we found that the generated code looked as a straightforward compiler would generate for:

```
L_skip:
  ... skipping ...
  code snippet in C;
  threaded-code dispatch;
L_before:
  code snippet in C;
  threaded-code dispatch;
L_after:
  threaded-code dispatch;
```

I.e., gcc-7 duplicated code reached by jumping to a label and the same code being reached in a straight-line way. This may be a useful optimization, but it means that our code snippets now contain the dispatch code, which is contrary to our intentions.

We found the following workaround: In order to convince gcc that this code duplication does not pay off, after each label we insert 8 `asm` statements, each containing a comment with a text unique to that label (so gcc hopefully will not try to deduplicate the code). Currently this is enough to convince gcc to avoid the code duplication

5.5 Register allocation

Virtual machines have a number of “registers”, which are implemented in C code as C (local) variables. At least for the frequently-used variables, it would help performance if they were allocated to real-machine registers.

Up to and including gcc-9, we explicitly assigned registers to several of these variables on many platforms with GNU C's feature “Explicit Register Variables”. In gcc-10 and later, disabling the explicit register variables produced better results than enabling them.

With either approach, we have the following problem: In the Gforth engine, gcc only used callee-saved registers for these variables. With explicit register variables, because gcc does not accept caller-saved registers for those. But if left to itself, gcc does not use caller-saved variables, either, because `engine()` contains about 100 VM instruction implementations that perform calls, and these calls apparently cause the compiler to avoid using caller-saved registers for these variables, especially for those that are used in < 100 VM instructions, such as the return-stack pointer of Gforth. A problem here is that gcc does not know that VM instructions that access the return stack are used frequently, while VM instructions that perform calls tend to be used rarely. This is a problem even for architectures like Alpha that have a lot of registers in principle, but a calling convention with relatively few callee-saved registers.

For being able to use additional registers for stack caching without spilling other VM registers, we use the following observation: All VM instruction implementations that contain a call only use the canonical state with one stack item in a register, due to non-relocatability. So additional stack cache registers are dead at the end of these VM instruction implementations, and there is no reason to preserve these registers across the calls. But how do we tell gcc about that?

```
L_skip:
  ... skipping ...
L_before:
  code snippet containing a call;
  asm("":"=X"(spb));
  asm("":"=X"(spc));
L_after:
  threaded-code dispatch;
```

The empty `asm` statements right before `L_after` claim to overwrite `spb` and `sbc` (the variables holding the additional stack-cache items in some stack-cache states). Therefore, these variables are dead at the call and do not need to be preserved. This means that this VM instruction implementation is no hindrance to allocating `spb` and `spc` in a caller-saved register. And indeed, one of these variables is allocated by gcc in a caller-saved register.

Another way to influence the register allocator that we have not used is the GNU C extension “Label Attributes” (available since gcc-5). We can declare the VM instruction implementations with calls as being `cold`, and/or declare frequently-used VM instruction implementations to be `hot` by following the label with an attribute:

```
L_skip:
  ... skipping ...
L_before: __attribute__((cold));
  code snippet containing a call;
L_after:
  threaded-code dispatch;
```

With that, the register allocator is hopefully more willing to use caller-saved registers for local variables of the VM.

5.6 Cache consistency

Many architectures do not guarantee cache consistency between data and instruction caches, and require a special piece of code between generating code and executing code; this incantation typically consists of a few lines of architecture-specific (or, on some architectures worse, implementation-specific or OS-specific) code, and for a long time has been the only non-portable part of Gforth's code copying implementation. Gcc-4.3 introduced `__builtin___clear_cache()`, which would eliminate this last piece of non-portability. We use `__builtin___clear_cache()` on RISC-V.

Unfortunately, `__builtin___clear_cache()` is not implemented correctly on at least PowerPC64.[13] We have switched Gforth back to using architecture-specific implementations of this functionality (except on RISC-V). When implementing your own code-copying compiler, check if `__builtin___clear_cache()` is compiled to non-empty code on each architecture that requires special code to make the caches consistent. If it compiles to non-empty code, that code will hopefully be correct.

Another problem with such architectures is multithreading: The code-generating thread must ensure that the D-cache lines are written to a common memory, and then the code-executing threads must invalidate these regions in the I-cache (to get rid of stale I-cache lines); due to prefetching and branch prediction, this may even be necessary if code in the address range has never been executed.

Until now we have ignored this problem, and relied on our luck. Typically Gforth programs only start subthreads after finishing compiling the source code (and thus code generation), which may explain why we have not seen any problems from that. A system with on-demand code generation (the narrow meaning of JIT) may be more likely to encounter such problems, however.

5.7 Spectre

GCC offers mitigations against Spectre v2 [KHF+19]. While all of these mitigations are expensive, because they disable indirect-branch prediction, the option `-mindirect-branch=thunk-inline` is less expensive than `-mindirect-branch=thunk`, because the latter makes the code snippets non-relocatable, so every VM instruction performs an indirect branch, while with the former option the relocatability of the code snippets is not affected, resulting in fewer indirect branches and therefore less slowdown.

On a Ryzen 3900X, we see slowdowns by a factor of 2.1–7.6 from using `-mindirect-branch=thunk-inline` and slowdown factors of 7.5–18.1 from using `-mindirect-branch=thunk`.

However, if you want to implement your programming language with Spectre mitigations, you

[13] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=93811

will prefer approaches such as copy-and-patch compilation that avoid performing so many indirect branches. You will also want to use mitigations against other Spectre vulnerabilities (e.g., speculative load hardening [ZBC$^+$23] against Spectre v1), which will introduce additional slowdowns for any approach, but unfortunately, these other mitigations require more work than just setting a C compiler flag.

5.8 Control-flow protection

There are exploit techniques such as return-oriented and jump-oriented programming that work by returning or jumping to arbitrary code. To make it more difficult to use these techniques, architectures and compilers offer ways to check that branches and returns only jump to targets that the compiler had in mind. E.g., gcc with the option `-fcf-protection=full` inserts an `endbr64` instruction at every indirect-branch target (i.e., every label in `engine()`), and the CPU can be told to report an error on an indirect branch to some other code. `Endbr64` is an AMD64 instruction, some other architectures have similar features.

This works with code copying: It copies the `endbr64` instruction to those places that the dispatch code will later indirect-branch to (and to additional places).

We use `-fcf-protection=none` in Gforth, however, because Gforth offers enough gadgets[14] already at the intended targets of indirect branches: All the VM instructios; moreover, Gforth and its VM is a low-level language that allows arbitrary memory access within the process. So a Gforth program that is exposed to untrusted input has to successfully defend against an attacker at the front line (source-level bounds checks etc.) and cannot make life harder for the attacker who has breached the front-line defense.

However, if your language is better suited to defense-in-depth, you can enable `-fcf-protection=full`, and they will work with code copying. This feature may cost a little performance, though: All the `endbr64` instructions need to be decoded and executed. In a small experiment with Gforth on a Ryzen 8700G (Zen4), we saw an increase in instruction count by a factor 1.45 and an increase in cycle count by a factor 1.04 from `-fcf-protection=full`. Narrower processors may see a bigger slowdown (the instructions per cycle on Zen4 increased from 3.83 to 5.34). VM implementations with more machine instructions per VM instruction will see a smaller effect.

5.9 Clang

Clang supports "Labels as Values", and Gforth is built with clang on platforms where GCC is not available. However, using Clang poses a number of problems:

- Clang wants to understand the assembly language in `asm` statements, and stops compiling when it sees `asm("SKIP4")`. One can work around that, and that is done in the ports that need clang, but we have not done that for the experiments on Debian Linux in the following.
- Clang takes much longer than gcc to compile Gforth's `engine()` and also needs more memory. As an example, for `gforth-itc` (an indirect-threaded-code Gforth without code copying nor other optimizations, and therefore without `SKIP4`), on a Ryzen 5800X gcc-12.2 takes 3s and 346MB to compile `engine()`, while clang-14.0.6 takes 699s and 5603MB. For `engine()` for `gforth-fast` (with all optimizations enabled), clang takes 3399s and 18264MB before it stops compiling because of `SKIP4` (gcc takes 26s and 1804MB).
- Clang generates a lot of register and memory shuffling code, similar to what we have seen with gcc-3.0. As a result, runnung the small benchmarks on Clang-compiled `gforth-itc` executes 6.4 times more AMD64 instructions than on GCC-compiled `gforth-itc` and consumes 4.2 times more Ryzen 5800X cycles.

As a result, Gforth selects GCC whenever it can. We expect that the clang compilation speed will be a problem for other code-copying compilers. The bad code generation may be less pronounced in language implementations that rely less on copy propagation than Gforth. Clang may be more viable when using tail calls instead of using one function and "Labels as Values" (see Section 7.1).

6 OS issues

Over the years operating systems have restricted executing dynamically-generated code more and more. In the beginning, all memory was allocated with read, write, and execute (RWX) permissions; later, `malloc()` only allocated RW memory, and one has to use `mmap()` to get RWX memory.

Recently, some operating systems (in partcular MacOS on Apple silicon) do not serve `mmap()` calls that ask for RWX memory (this restriction is also known as `W^X`). This is a problem for all systems with run-time code generation, not just code-copying compilers, but, e.g., Java JITs as well. For a single-threaded language implementation, one can

[14] In the context of return-oriented and jump-oriented programming, a gadget is a machine-code sequence that an attacker may want to return/jump to.

mprotect() the memory to W when generating the code, and to X when executing it, but that does not work for multi-threaded code, unless you want to start a new page whenever you generate a new piece of code.

MacOS provides a MacOS-specific API for JIT compilers that supports switching the memory into W in the code-generating thread and keeping it X in the other threads, and Bernd Paysan has actually invested the time to use this API.

Several of the BSDs also has W^X by default, but allows to mark binaries such that RWX works. The command for marking the binary is short, but specific to the BSD variant.[15]

An approach that may work without special APIs is to have the code generation in one process and the execution in a different process, both mapping the same memory, but with different permissions. Another option may be to map the same memory within one process twice, at one address range with W permission, and at the other address range with X permission. We have not tried either approach.

If all else fails or you don't want to jump through the hoops that these operating systems put up, code-copying based on threaded code always allows you to fall back to plain threaded code, which works fine on operating systems with the W^X restriction. E.g., Gforth-0.7 (which was not specifically designed for this circumstance) automatically falls back to plain threaded code on MacOS on Apple silicon: the mmap() call for allocating the code memory fails, so Gforth-0.7 falls back to using malloc(), and because that does not produce executable memory on modern OSs, Gforth-0.7 turns off dynamic code generation.

7 Alternative approaches

In this section we describe approaches that are interesting but that are not implemented in production Gforth.

7.1 Tail calls

Instead of putting all VM-instruction implementations in one function and using goto * for threaded-code dispatch, one can also put each VM instruction implementation in a separate function and use optimized tail-calls for threaded-code dispatch, as follows:

```
typedef void (*vm_inst)(void **ip,
                        long *dsp, long tos);

void lit(void **ip, long *dsp, long tos)
{
  ... payload including ip update ...;
  (*(((vm_inst *)ip)[0]))(ip,dsp,tos);
}
```

The last line of the function performs the threaded-code dispatch. The tail-call must be optimized into a jump, otherwise the C stack grows and eventually overflows. When we first considered this approach [Ert95], GCC did not tail-call optimize such code, but in the meantime it does, as does Clang [XK21]; Clang even provides a way to require that a call is tail-call-optimized, and will report an error if it cannot meet this requirement.

The VM registers are passed as parameters, at least as long as the calling convention supports passing them in machine registers. With gcc, additional VM registers could be stored in global explicit register variables; on AMD64 this results in 12 general-purpose and 8 floating-point registers available for VM registers. Clang does not support explicit register variables, but it supports using a calling convention for these functions and calls that uses as many registers as possible for parameter-passing.

So for dealing with VM registers efficiently, one has to pass VM-registers in parameters or keep them in global register variables with compiler-dependent and ABI-dependent code, but that is a relatively small effort.

With the tail-calling approach, there is a fixed allocation of VM registers to machine registers, either coming from the position in the parameter list, or from the explicit register allocation.

We expect that the VM instruction implementations can be compiled faster and with less memory with the tail-calling approach, because the compiler will hopefully not try to perform data-flow analysis between the functions, while it tries to do it when the implementations are all contained in one function. We can then squander the compilation speed gain on introducing more code snippets, for various optimization purposes (Xu and Kjolstad report using 98831 code snippets [XK21]).

Another benefit is that we should see no or little of the register-and-memory shuffling that we see with Clang, or with gcc without -fno-tree-vectorize.

So far you have only seen how tail calls can be used to implement threaded code. How can it be used for code-copying compilation?

In order to do that, we need a way to get rid of the dispatch part of the implementation. Unfortu-

[15] https://www.reddit.com/r/BSD/comments/10isrl3/notes_about_mmap_mprotect_and_wx_on_different_bsd/

nately, compilers tend to mix the instructions from the payload part with those from the dispatch part; just inserting a label between them will not work, because there is nothing that jumps to this label. Maybe an `asm` statement can be made to act as a barrier, but preliminary experiments failed to produce satisfying results.

One way that may be more promising is to have, in addition to functions that end in a threaded-code dispatch (to have a fallback option), variants intended only for code copying that end in a direct [XK21]) or indirect tail-call without threaded-code dispatch. On many architectures this is just one instruction, that must be last in the function. However, there are exceptions: Some architectures have delayed branches (HPPA, MIPS, SPARC); some architectures require two instructions for indirect branches (PowerPC, IA-64). In some programming models, a direct jump to a function is expressed as an indirect jump to a target loaded from the global offset table (GOT), and as a result the direct jump also is expressed with more than one instruction.

Once we have solved the problem of keeping the payload separate from the tail call, how do we know where the tail call starts so that we can use the code between the start of the function and this instruction as code snippet? Xu and Kjolstad extract the function size (and the code) from the object file (see Section 7.2), and apparently use their own architecture-specific knowledge about the size of the last instruction to determine where it starts. A way to determine the size of this last instruction may be to have a function that performs only this tail-call, and look at its size.

7.2 Snippets from object files

Gforth extracts code snipets from the executable at run-time and has some startup overhead while it examines all the code snippets for relocatability and performs its table setup.

An alternative is to extract code snippets from object files [NHCL98, XK21] at system build time using the Binary File Descriptor library (GNU BFD). One advantage of this approach is that the object file contains additional information, such as the function size, or linkage information for symbols external to the object file.

7.3 Copy-and-patch compilation

Gforth accesses immediate operands and control-flow information through IP. This requires a register for IP, results in less efficient accesses to immediate operands and less efficient control flow than with ordinary compilers, and requires keeping the VM code around.

An alternative is to have code snippets that contain dummy immediate arguments and perform control flow directly to dummy targets, and then patch the constants or target addresses in these code snippets with the actual values, resulting in copy-and-patch compilation.

One approach for copy-and-patch compilation has been based on using the linkage information in object files [NHCL98, TCL$^+$00, XK21]. References to external symbols are used for patchable immediate operands and patchable control-flow targets. The linkage information describes where to patch and how to patch (e.g., absolute or relative address). This requires some architecture/ABI-specific work, but ABIs have a finite number of relocation types (e.g., 52 in the AMD64 ABI [LMG$^+$]) and only a few are actually used in the code snippets.

However, by refering to an external symbol the copy-and-patch compiler usually cannot patch the immediate operand of instructions like RISC-V's `addi`. The external symbol is a 64-bit (or 32-bit) value, while the immediate operand of `addi` is 12 bits long, so the addition of a constant (whatever its size) is compiled to several instructions.

Another approach is to start with code snippets delimited by labels in one C function, like Gforth's code copying uses, but perform patching in addition [VA04, EG04b].

We implemented copy-and-patch compilation for Gforth in a prototype for IA-32 and PowerPC using the latter approach [EG04b]. This work was based on Gforth's approach of extracting code snippets from the executable at system startup time. The `engine()` function was compiled thrice, twice with the same immediate arguments, and once with different immediate arguments. The first two versions were compared to determine relocatability, the third version was compared to find out the placeholders of the immediate arguments.

This approach can make use of the RISC-V `addi` instruction, but needs to fall back to code that uses several instructions when the immediate operand becomes too large. It needs quite a bit of knowledge about the instruction encodings, in particular, the sizes of the immediate-operand fields. We considered determining the encoding and size by varying the immediate operands a lot more, but did not implement that idea; dealing with each architecture manually is probably less work.

We originally intended to turn this copy-and-patch compiler into a production engine for Gforth, but in those years several GCC releases resulted in falling back to threaded code, so the copy-and-patch approach looked too brittle, and we let it bitrot. Later, the rethoric by the advocates of C code without undefined behaviour kept the distrust in GCC high. If we had continued to maintain this engine, maybe we could now report on its success and the hurdles we had to overcome. Or maybe it would have been a bridge to far.

8 Related Work

GCC-2.0 (released February 1992) introduced "Labels as Values", which not only proved useful for implementing threaded code (we started the Gforth project[Ert93] in July 1992), but also for compiling by copying compiler-generated code snippets between two labels, with all the code snippets being within a function. This method was first outlined by Rossi and Sivalingam [RS96, Section 2.5], who refer to an unpublished discussion between Xavier Leroy and Kenneth Oksanen. Piumarta and Riccardi provided a more elaborate treatment [PR98], with deduplication of code sequences.

Ertl and Gregg implemented code-copying in Gforth, and in the beginning the main benefit was in indirect branch prediction accuracy [EG03a, EG03b, CEG07]; it turned out that leaving away deduplication (or conversely, introducing replication, as we framed it) helped the branch predictors at the time. Indirect branch predictors have improved a lot in general-purpose processors [RSS15], but code copying still provides a good speedup.[16]

Once you have code copying, you can eliminate instruction-pointer (IP) updates, either by leaving away the unneeded VM instruction slots [PR98], or by replacing several IP updates with a combined one [EP24]. While IP updates play a minor role for performance on CPUs from the 2000s, they can be the decisive bottleneck on loop-dominated benchmarks in the 2020s.

Another optimization that was facilitated by code copying is multi-state stack caching [Ert95, EG04a, EG05].

Tempo is a partial evaluator that uses code copying and patching by extracting information from object files [NHCL98]; Tempo was later used to specialize an interpreter into a compiler [TCL$^+$00].

Iliasov [Ili03] describes a copy-and-patch compiler with a minimal patching component: Only literals need to be patched; control flow is performed by performing indirect jumps to addresses provided as literals.

QEMU is a full-system emulator. It is a production system with a long history, and has many more users than Gforth. QEMU can emulate machines with a different instruction set than the host machine. It uses dynamic translation techniques for that, originally implemented in its Dyngen component [Bel05] using code-copying and patching, similar to what we described in Section 7.2 and 7.3. But Dyngen uses ordinary functions, not tail-calling functions, and has to get rid of the function prologue and epilogue. Dyngen is gcc-3.x-specific, and it apparently was too difficult to adapt it to newer gcc versions or other compilers, so it was replaced with TCG in QEMU-0.10.0 released in 2009. TCG is based on QOP by Paul Brook, who described it as "Hand written code generator"[17], so TCG probably is not based on copying and pasting compiler-generated code.

In Gforth we have dealt with changes in GCC by finding workarounds, or, for versions where we were not successful, by falling back to threaded code. Another approach is to actually define the properties that a compiler's code generation should have to support code copying; then modify a compiler to provide those properties (when asked for it), and report an error if it fails to provide the properties. This approach has been explored by Prokopski and Verbrugge [PV07, PV08], but their patches have not been integrated into GCC.

Several code-copying JavaVM implementations have been implemented, among them SableVM [GH03] and the Cacao interpreter [ETK06]. A particular challenge solved by these implementations was quickening of VM instructions, where VM instructions rewrite themselves into faster code on first execution. SableVM stopped being maintained after the research project ended (last release 2007). The Cacao interpreter bit-rotted while the main thrust of Cacao continued to use conventional code generation technology.

Maxine is a Java VM implementation with two-level compilation (baseline and optimizing compiler), where the baseline compiler is a copy-and-patch compiler that uses templates written in Java and where the code is generated by the optimizing compiler (which uses conventional compiler techniques) or by HotSpot [WHV$^+$13].

Xu and Kjolstad implement two copy-and-patch compilers: One that directly compiles from the abstract syntax tree (AST) without going through a VM and one for WebAssembly. Their technique works by having each code snippet (called stencil in the paper) in a tail-calling function with references to external symbols as placeholders for patching, and extracting the code snippets from object files. They use 1666 code snippets for the WebAssembly compiler, and 98831 code snippets for the AST compiler; the latter is notable, because it is beyond practical for the technique where all code snippets are in one function.

[16] See Section 2.1 and http://www.complang.tuwien.ac.at/anton/interpreter-branch-pred.txt.

[17] https://qemu-devel.nongnu.narkive.com/bCtjCaPs/hand-written-code-generator-2

9 Conclusion

Code-copying compilers make retargeting of the compiler much easier by using code snippets coming from a different compiler. Gforth demonstrates that code-copying without patching can produce code with similar performance as a compiler with a hand-written architecture-specific code generator. Gforth has used code copying since 2003, on many architectures, and has dealt with many GCC versions in those years. If all else fails, Gforth can fall back to threaded code, but it usually does not have to.

Copy-and-patch compilation promise an improvement in performance over copying without patching (as in Gforth) at a moderate increase in architecture-specific code. However, while there have been a number of publications about this technology, no production system is known to us that currently uses it.

References

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973. 3.1

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Freenix Track of Usenix Annual Technical Conference*, pages 41–46, 2005. 8

[CEG07] Kevin Casey, M. Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Transactions on Programming Languages and Systems*, 29(6):37:1–37:36, October 2007. 8

[EG03a] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, 2003. 3.2, 8

[EG03b] M. Anton Ertl and David Gregg. The structure and performance of *Efficient* interpreters. *The Journal of Instruction-Level Parallelism*, 5, November 2003. http://www.jilp.org/vol5/. 8

[EG04a] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Interpreters, Virtual Machines and Emulators (IVME '04)*, pages 7–14, 2004. 2.1, 3.3, 8

[EG04b] M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004. 7.3

[EG05] M. Anton Ertl and David Gregg. Stack caching in Forth. In M. Anton Ertl, editor, *21st EuroForth Conference*, pages 6–15, 2005. 2.1, 8

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002. 2.1

[EP24] M. Anton Ertl and Bernd Paysan. The Performance Effects of Virtual-Machine Instruction Pointer Updates. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:26, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 2.1, 3.3, 8

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993. 8

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 315–327, 1995. 7.1, 8

[Ert14] M. Anton Ertl. How to get rid of C. In *30th EuroForth Conference*, pages 63–65, 2014. 5

[Ert24] M. Anton Ertl. Interpreter vs. compiler performance at run-time. In *Tagungsband des Jahrestreffens 2024 der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte"*, INSIGHTS — Schriftenreihe der Fakultät Technik, pages 7–12, 2024. 2.1

[ETK06] M. Anton Ertl, Christian Thalinger, and Andreas Krall. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4:25–32, 2006. Journal papers from *.NET Technologies 2006* conference. 8

[GH03] Etienne Gagnon and Laurie Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. In *Compiler Construction (CC '03)*, volume 2622 of *LNCS*, pages 170–184. Springer, 2003. 8

[Ili03] Alex Iliasov. Templates-based portable just-in-time compiler. *SIGPLAN Notices*, 38(8):37–43, August 2003. 8

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019. 5.7

[LMG+] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell, editors. *System V Application Binary Interface — AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models)*. 7.3

[NHCL98] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimantal study. In *IEEE International Conference on Computer Languages (ICCL '98)*, pages 123–142, 1998. 7.2, 7.3, 8

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998. 3.2, 8

[PV07] Gregory B. Prokopski and Clark Verbrugge. Towards GCC as a compiler for multiple VMs. In *Proceedings of the GCC Developers' Summit*, pages 117–129, 2007. 8

[PV08] Gregory B. Prokopski and Clark Verbrugge. Compiler-guaranteed safety in code-copying virtual machines. In *Compiler Construction (CC'08)*, pages 163–177. Springer LNCS 4959, 2008. 8

[RS96] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996. 3.2, 8

[RSS15] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters — don't trust folklore. In *Code Generation and Optimization (CGO)*, 2015. 8

[TCL+00] Scott Thibault, Charles Consel, Julia L. Lawall, Renaud Marlet, and Gilles Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, September 2000. 7.3, 8

[VA04] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In *IVME '04 Proceedings*, pages 42–50, 2004. 3.4, 7.3

[WHV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization*, 9(4):30:1–30:24, January 2013. 8

[XK21] Haoran Xu and Fredrik Kjolstad. Copy-and-patch compilation. *Proc. ACM Program. Lang.*, 5(OOPSLA):136:1–136:30, October 2021. 7.1, 7.2, 7.3

[ZBC+23] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: Taking speculative load hardening to the next level. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7125–7142, Anaheim, CA, August 2023. USENIX Association. 5.7

Stephen Pelc, stephen@vfxforth.com
Wodni & Pelc GmbH
9 September 2025

Forth for ARM 64 CPUs

Trials and tribulations

ARM's 64 bit CPUs are very different beasts from the 32 bit ones. The instruction mix Is completely different and appears to be derived from the Power PC. Cache behaviour Is quite different and poorly documented. VFX Forth (64 bit) has been ported to the ARMv8-A architecture and an alpha release is expected later in 2025. The current Arm® Architecture Reference Manual for the A profile is 14,568 pages long. Despite my reservations, the more I use this CPU, the more I come to appreciate it. The code density is far better than expected.

In the beginning …

Many years ago, the 32 bit ARM CPU appeared. Over the years various instruction sets and derivatives appeared. As a company, ARM has always made frequent changes to the instruction set to support the silicon. Unlike other companies, an ARM instruction set supports today's architecture. 64 bit ARMs are based on the ARMv8 architecture from 2011, and ARMv9 already exists. ARMv9 is basically has the ARMv8 instruction set with extensions.

Under some customer pressure, we started a port of VFX Forth to the ARMv8A architecture.

About the ARM64

The ARM 64 bit CPU in native mode has very little to do with ARM32, although the original ARM32 and Thumb-2 ISAs are supported for legacy reasons and I shall discuss these no further. There are several ARM64 assembler books out there; in the main they are useless for people who have assembler experience and projects under their belt. You will find the

“Compiler Writer’s Guide to the Power PC” of use, and PowerPC assembler experience of value.

The instruction set consists of 32 bit instructions only, with a number of conditional operations derived from PowerPC. Many of the instructions have three operands. There are 32 registers, including a zero register and a subroutine return register. There are many pseudo-instructions which offer a different syntax to the base instruction. Although initially confusing, the instruction set makes sense after a while. Despite this, a description of the **BFM**, **SBFM** and **UBFM** instructions for human beings would be very useful.

The big change is in the cache architecture, which is a real pain. We have not finished with it yet.

Instruction set

I will only discuss the basic and integer portions of the instruction set here. There is a full set of basic instructions plus enough special instructions for supporting cryptography, security and ARM32 that one could already call the instruction set baroque.

Poor code density has been a problem for many RISC or load/store architectures. A side effect of improving code density for ARM64 has been a selection of immediate value encodings -

1) Arithmetic - 16 bit, 12 bit+shift, 12 bit, 8 bit, 7 bit, 6 bit.
2) Logical - 13 bit mask about 5000 options. Used by **AND**, **ORR** and **EOR**.
3) Branch offsets - 26 bits, 19 bits, 14 bits
4) Memory offsets - s19 bits, u12 bits, s9 bits, s7 bits, 0

The branch instructions result in a call range of +/-128 Mbytes. Most conditional branches have a +/-1 Mbytes branch range. This is a vast improvement over many other CPUs.

Because of the impact of mispredicted branches on performance, conditional instructions reduce both code size and improve performance by avoiding conditional branches, e.g. the end of **WITHIN**

```
cmp      tos, x17            \ (n1-n2)-(n3-n2)
csinv    tos, xzr, xzr, .ls  \ cy -> -1, ncy -> 0
```

Conditional select invert

This instruction returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register. See: **CSEL**, **CSET**, **CSETM**, **CSINC**, **CSINV**, and **CSNEG**.

Caches

The cache system for ARM64 is quite different to that for ARM32. There L1 caches for both code and data. The minimum cache line size is 64 bytes, but is permitted to be larger. The size can be read by application programs running at execution level 0 (EL0). A limited range of cache maintenance instructions can be run at EL0 that permit the cache to be flushed by code running at EL0.

Tools

The main tools we use for porting are the VFX cross compiler. Testing is performed on a 64 bit version of ARM linux. We have used both UTM and Parallels to host these on Apple Silicon Macs. Apple produce a tool called Rosetta that enables x64 applications to run on Apple Silicon. Two problems with Rosetta have forced us to abandon it.

1) x64 Forth cross compilers are slowed down by a factor of 100 or more.

2) There are bugs in Rosetta that have forced other projects to abandon it.

We have therefore moved back to cross-compiling on an x64 box, copying the output to an ARM Linux and then debugging.

For both the cross compiler and the target code there are five sections that change for each target

1) Assembler - an ARM64 assembler is provided with a prefix notation that closely follows the ARM64 standard notation.

2) Disassembler - it is almost impossible to debug compiled native code without one.

3) Code generator and optimiser - produces faster and shorter code than that produced by combining patterns. The VFX code generator is an analytical compiler that tracks which registers are used and what they contain.

4) Required code - these are words that either cannot be written easily in high level Forth, or should be written in assembler for performance reasons.

5) Operating system interface - calling functions in shared libraries and providing callbacks that can be used by the operating system.

The following are examples of these.

```
code l!c(t) \ l addr --
     \ *G Store and Flush instruction cache line containing *\i{addr}.
     \ ** Use in stores of code.
       ldr   x0, [ psp ]!, # 8        \ get l
       str   w0, [ tos, # 0 ]         \ save l
       ic    ivau tos                 \ flush
       dsb    SY                      \ ensure completion of the invalidation
       isb    SY                      \ ensure instruction fetch path sees
                                      \ new I cache state
       ldr    tos, [ psp ]!, # 8      \ restore TOS
       ret   x30
     end-code
```

```
dis l!c(t)
L!C(T)
( 0041:1850 A08740F8 )              LDR       X0, [ XPSP  ]!, # $8
( 0041:1854 400300B9 )              STR       W0, [ XTOS, # $0 ]
( 0041:1858 3A750BD5 )              SYS       $1BA9 XTOS
( 0041:185C 9F3F03D5 )              DSB       # $0F
( 0041:1860 DF3F03D5 )              ISB       # $0F
( 0041:1864 BA8740F8 )              LDR       XTOS, [ XPSP  ]!, # $8
( 0041:1868 C0035FD6 )              RET       XLR  ( NEXT/EXIT )
28 bytes, 7 instructions.
```

```
: InOvl?      \ addr1 -- addr2|0
\ *G Returns the overlay address (addr2) if the address (addr1)
\ ** is within an overlay, otherwise returns 0.
  ovl-link @
  begin                         \ -- addr *ovl
    dup
  while                         \ -- addr *ovl
    2dup OVI.end 2@ within if   \ -- addr *ovl
      nip  exit
    then
    ovi.link @
  repeat
  nip                           \ remove addr
;
```

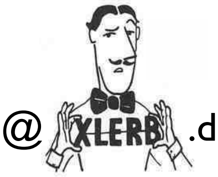
```
dis inovl?
INOVL?
( 0042:9F88 00FF9AD2 )              MOVZ      X0, # $D7F8
( 0042:9F8C 2008A0F2 )              MOVK      X0, # $41 LSL # $10
( 0042:9F90 110040F8 )              LDUR      X17, [ X0, # $0 ]
( 0042:9F94 BA8F1FF8 )              STR       XTOS, [ XPSP, # $-8 ]!
( 0042:9F98 FA0311AA )              MOV       XTOS, X17
( 0042:9F9C 9E8F1FF8 )              STR       XLR, [ XRSP, # $-8 ]!
( 0042:9FA0 5F031FEB )              CMP       XTOS, XZR  LSL# $00
( 0042:9FA4 40020054 )              B .EQ     # $429FEC
( 0042:9FA8 50BF41A9 )              LDP    X16, X15, [ XTOS, # $18 ]
( 0042:9FAC BD6300D1 )              SUB       XPSP, XPSP, # $18
( 0042:9FB0 AF0300F9 )              STR       X15, [ XPSP, # $0 ]
( 0042:9FB4 A10F40F9 )              LDR       X1, [ XPSP, # $18 ]
( 0042:9FB8 A10700F9 )              STR       X1, [ XPSP, # $8 ]
( 0042:9FBC BA0B00F9 )              STR       XTOS, [ XPSP, # $10 ]
( 0042:9FC0 FA0310AA )              MOV       XTOS, X16
( 0042:9FC4 53A3FF97 )              BL        # $412D10     WITHIN
( 0042:9FC8 5F031FEB )              CMP       XTOS, XZR  LSL# $00
( 0042:9FCC BA8740F8 )              LDR       XTOS, [ XPSP  ]!, # $8
( 0042:9FD0 80000054 )              B .EQ     # $429FE0
( 0042:9FD4 BD230091 )              ADD       XPSP, XPSP, # $08
( 0042:9FD8 9E8740F8 )              LDR       XLR, [ XRSP  ]!, # $8
( 0042:9FDC C0035FD6 )              RET       XLR  ( NEXT/EXIT )
( 0042:9FE0 510340F8 )              LDUR      X17, [ XTOS, # $0 ]
( 0042:9FE4 FA0311AA )              MOV       XTOS, X17
( 0042:9FE8 CEFDFF54 )              B         # $429FA0
( 0042:9FEC BD230091 )              ADD       XPSP, XPSP, # $08
( 0042:9FF0 9E8740F8 )              LDR       XLR, [ XRSP  ]!, # $8
( 0042:9FF4 C0035FD6 )              RET       XLR  ( NEXT/EXIT )
112 bytes, 28 instructions.
 ok
```

Blending Forth

mixing other languages and Forth

EuroForth'25 conference 2025-09

Ulrich Hoffmann

uho@.de

Overview

- introduction
- implementing Forth in other languages
- abstraction and representation
- blending Forth
- demo
- conclusion

Day of the Week and Zeller's congruence

$$h = \left(q + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor - 2J\right) \bmod 7,$$

```
function ZellerDayOfWeek(q, m, y: Integer): Integer;
…
begin

  K := y mod 100;   // year of the century
  J := y div 100;   // zero-based century

  h := (q + ((13 * (m + 1)) div 5) + K +
                           (K div 4) + (J div 4) - (2 * J)) mod 7;
  …
  ZellerDayOfWeek := h
end;
```

Day of the Week and Zeller’s congruence

$$h = \left(q + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor - 2J\right) \bmod 7,$$

In **standard Pascal (ISO 7185 and its descendants like Free Pascal, Turbo Pascal, Delphi)** the `mod` operator always returns a result with the **same sign as the dividend** (the left operand).

```
Writeln(  7 mod 3 );    // 1
Writeln( -7 mod 3 );    // -1
Writeln(  7 mod -3 );   // 1
Writeln( -7 mod -3 );   // -1
```

Day of the Week and Zeller’s congruence

$$h = \left(q + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor - 2J\right) \bmod 7,$$

```
function ZellerDayOfWeek(q, m, y: Integer): Integer;
…
begin

  K := y mod 100;   // year of the century
  J := y div 100;   // zero-based century

  h := (q + ((13 * (m + 1)) div 5) + K +
                           (K div 4) + (J div 4) - (2 * J)) mod 7;
  …
  ZellerDayOfWeek := h
end;
```

Day of the Week and Zeller’s congruence

$$h = \left(q + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor - 2J\right) \bmod 7,$$

```
function ZellerDayOfWeek(q, m, y: Integer): Integer;
…
begin

  K := y mod 100;   // year of the century
  J := y div 100;   // zero-based century

  h := (q + ((13 * (m + 1)) div 5) + K +
                           (K div 4) + (J div 4) + (5 * J)) mod 7;
  …
  ZellerDayOfWeek := h
end;
```

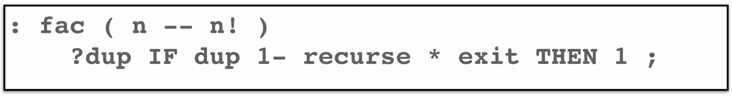
Implementing Forth in Python

- ongoing adventure to implement Forth in different languages
  - Assembler
  - Forth itself
  - Emacs-Lisp
  - Golang
  - Python
- Insightful discoveries

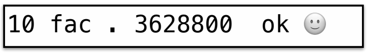
Implementing Forth in Python

- How to implemement stack and return-stack?
- How primitives?
- How the dictionary?
- How the inner and out interpreter?
- What about BASE and STATE?
- How to read characters one-by-one?

Factorial



```
: fac ( n -- n! )
    ?dup IF dup 1- recurse * exit THEN 1 ;
```

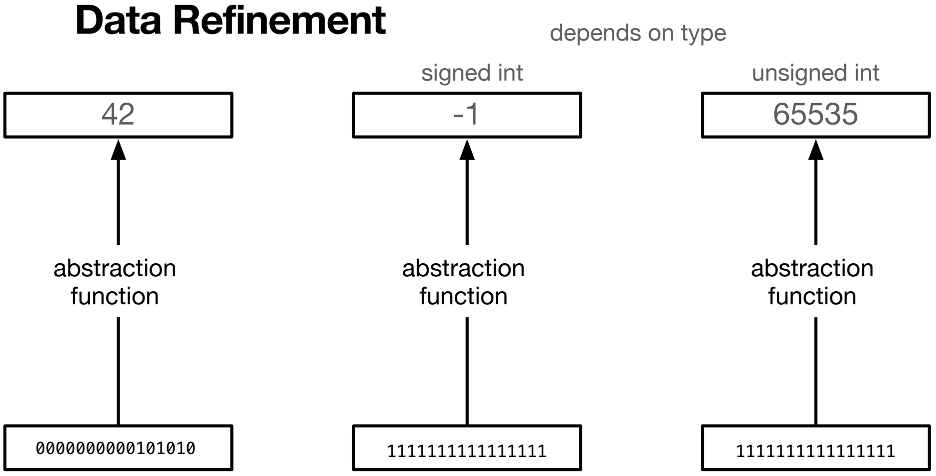


```
10 fac . 3628800  ok 🙂
```

Abstraction and Representation

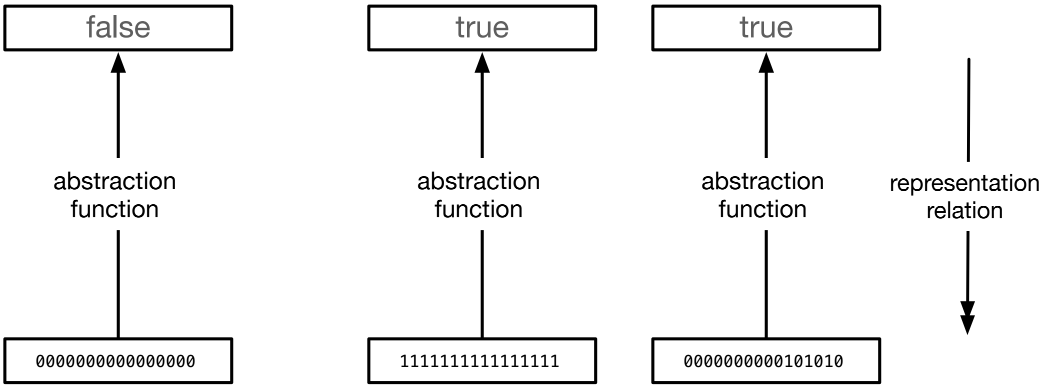
- You take the elements of the implementation language to realize the elements of the target language Forth.
- data refinement
- operator refinement

Data Refinement



abstraction function sometimes called *retrieval* function

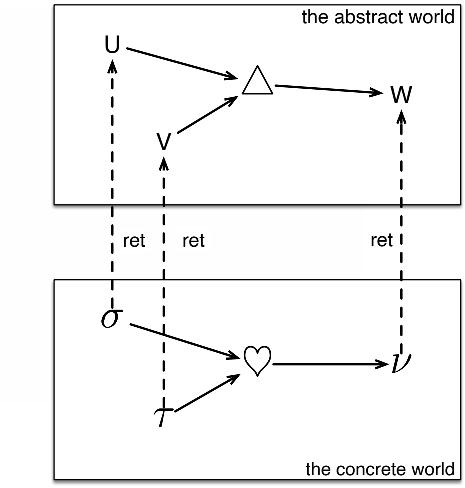
Data Refinement



representation relation sometimes called *refinement relation*

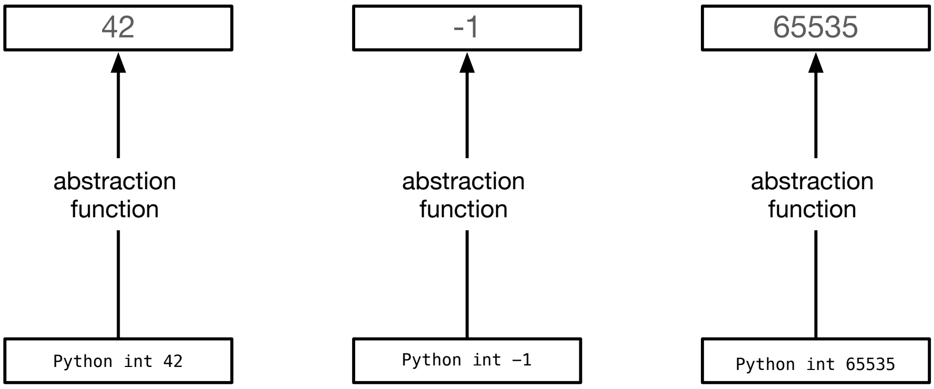
Operator Refinement

$$\mathit{ret}(\sigma \heartsuit \tau) = (\mathit{ret}\ \sigma) \triangle (\mathit{ret}\ \tau)$$



$$\begin{tikzcd}
(U, V) \arrow[r, "\triangle"] & W \\
(\sigma, \tau) \arrow[u, "ret"] \arrow[r, "\heartsuit"] & \nu \arrow[u, "ret"']
\end{tikzcd}$$

Data Refinement



42

-1

65535

abstraction function

abstraction function

abstraction function

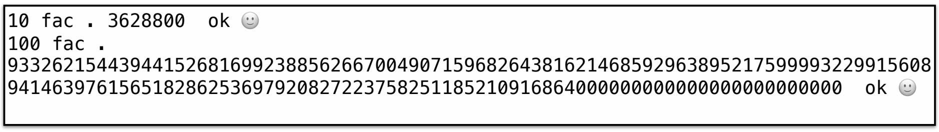
Python int 42

Python int -1

Python int 65535

Factorial

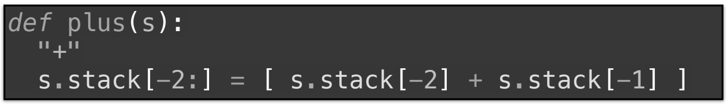
```
: fac ( n -- n! )
    ?dup IF dup 1- recurse * exit THEN 1 ;
```



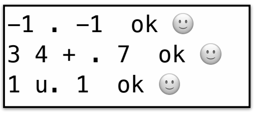
```
10 fac . 3628800  ok 🙂
100 fac .
93326215443944152681699238856266700490715968264381621468592963895217599993229915608
941463976156518286253697920827223758251185210916864000000000000000000000000  ok 🙂
```

Implementing Forth in Python

- Stack
- Primitives



```
def plus(s):
  "+"
  s.stack[-2:] = [ s.stack[-2] + s.stack[-1] ]
```

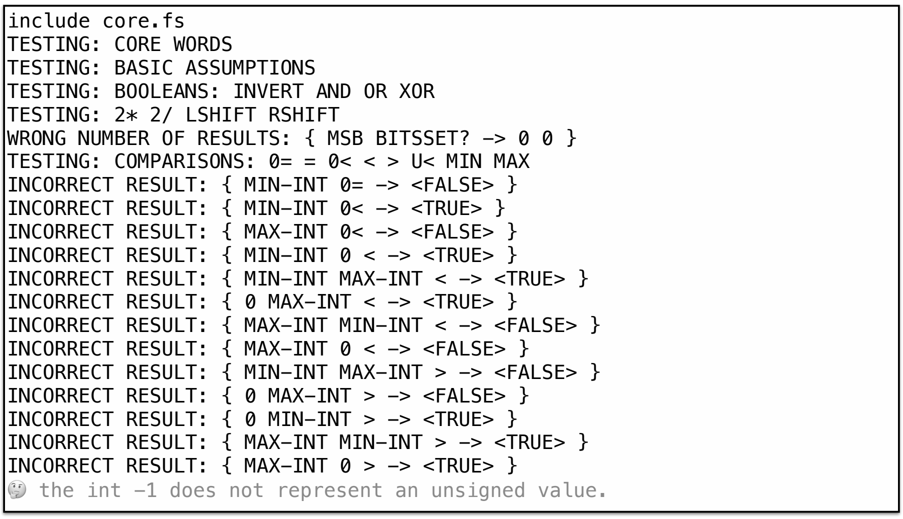


```
-1 . -1  ok 🙂
3 4 + . 7  ok 🙂
1 u. 1  ok 🙂
```

But is it Forth?

Let’s run the Forth-94 core test.

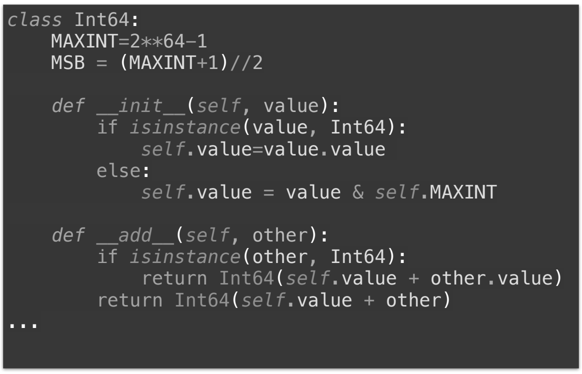
But is it Forth?



```
include core.fs
TESTING: CORE WORDS
TESTING: BASIC ASSUMPTIONS
TESTING: BOOLEANS: INVERT AND OR XOR
TESTING: 2* 2/ LSHIFT RSHIFT
WRONG NUMBER OF RESULTS: { MSB BITSSET? -> 0 0 }
TESTING: COMPARISONS: 0= = 0< < > U< MIN MAX
INCORRECT RESULT: { MIN-INT 0= -> <FALSE> }
INCORRECT RESULT: { MIN-INT 0< -> <TRUE> }
INCORRECT RESULT: { MAX-INT 0< -> <FALSE> }
INCORRECT RESULT: { MIN-INT 0 < -> <TRUE> }
INCORRECT RESULT: { MIN-INT MAX-INT < -> <TRUE> }
INCORRECT RESULT: { 0 MAX-INT < -> <TRUE> }
INCORRECT RESULT: { MAX-INT MIN-INT < -> <FALSE> }
INCORRECT RESULT: { MAX-INT 0 < -> <FALSE> }
INCORRECT RESULT: { MIN-INT MAX-INT > -> <FALSE> }
INCORRECT RESULT: { 0 MAX-INT > -> <FALSE> }
INCORRECT RESULT: { 0 MIN-INT > -> <TRUE> }
INCORRECT RESULT: { MAX-INT MIN-INT > -> <TRUE> }
INCORRECT RESULT: { MAX-INT 0 > -> <TRUE> }
🤔 the int -1 does not represent an unsigned value.
```

Implementing Forth in Python

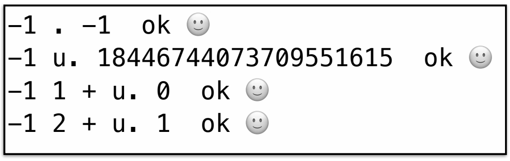
We need to implement cyclic 2’s complement numbers



```
class Int64:
    MAXINT=2**64-1
    MSB = (MAXINT+1)//2

    def __init__(self, value):
        if isinstance(value, Int64):
            self.value=value.value
        else:
            self.value = value & self.MAXINT

    def __add__(self, other):
        if isinstance(other, Int64):
            return Int64(self.value + other.value)
        return Int64(self.value + other)
...
```

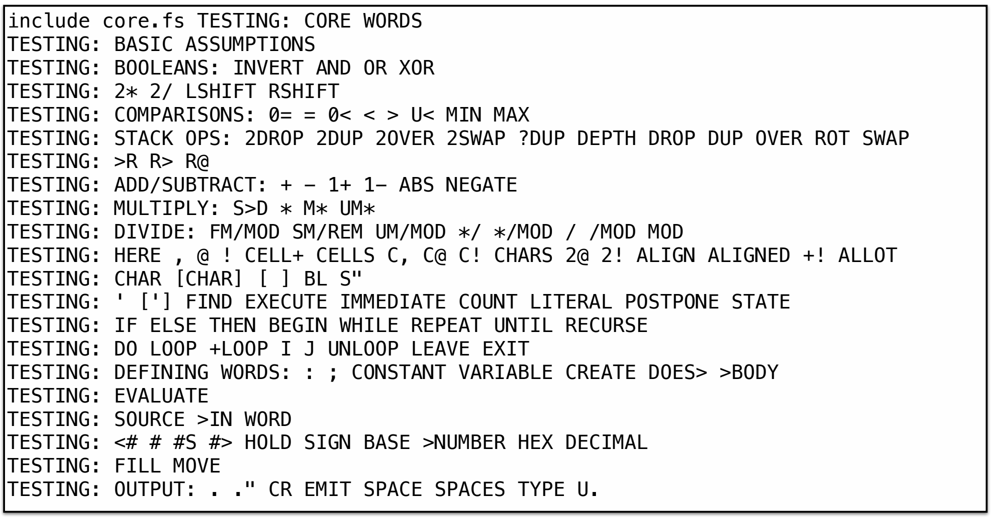


```
-1 . -1  ok 🙂
-1 u. 18446744073709551615  ok 🙂
-1 1 + u. 0  ok 🙂
-1 2 + u. 1  ok 🙂
```

But is it Forth?

Let’s run the Forth-94 core test - again.

But is it Forth?



```
include core.fs TESTING: CORE WORDS
TESTING: BASIC ASSUMPTIONS
TESTING: BOOLEANS: INVERT AND OR XOR
TESTING: 2* 2/ LSHIFT RSHIFT
TESTING: COMPARISONS: 0= = 0< < > U< MIN MAX
TESTING: STACK OPS: 2DROP 2DUP 2OVER 2SWAP ?DUP DEPTH DROP DUP OVER ROT SWAP
TESTING: >R R> R@
TESTING: ADD/SUBTRACT: + - 1+ 1- ABS NEGATE
TESTING: MULTIPLY: S>D * M* UM*
TESTING: DIVIDE: FM/MOD SM/REM UM/MOD */ */MOD / /MOD MOD
TESTING: HERE , @ ! CELL+ CELLS C, C@ C! CHARS 2@ 2! ALIGN ALIGNED +! ALLOT
TESTING: CHAR [CHAR] [ ] BL S"
TESTING: ' ['] FIND EXECUTE IMMEDIATE COUNT LITERAL POSTPONE STATE
TESTING: IF ELSE THEN BEGIN WHILE REPEAT UNTIL RECURSE
TESTING: DO LOOP +LOOP I J UNLOOP LEAVE EXIT
TESTING: DEFINING WORDS: : ; CONSTANT VARIABLE CREATE DOES> >BODY
TESTING: EVALUATE
TESTING: SOURCE >IN WORD
TESTING: <# # #S #> HOLD SIGN BASE >NUMBER HEX DECIMAL
TESTING: FILL MOVE
TESTING: OUTPUT: . ." CR EMIT SPACE SPACES TYPE U.
```

But is it Forth?

```
include core.fs TESTING: CORE WORDS
...
TESTING: OUTPUT: . ." CR EMIT SPACE SPACES TYPE U.
YOU SHOULD SEE THE STANDARD GRAPHIC CHARACTERS:
 !"#$%&'()*+,-./0123456789:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz{|}~
YOU SHOULD SEE 0-9 SEPARATED BY A SPACE:
0 1 2 3 4 5 6 7 8 9
YOU SHOULD SEE 0-9 (WITH NO SPACES):
0123456789
YOU SHOULD SEE A-G SEPARATED BY A SPACE:
A B C D E F G
YOU SHOULD SEE 0-5 SEPARATED BY TWO SPACES:
0  1  2  3  4  5
YOU SHOULD SEE TWO SEPARATE LINES:
LINE 1
LINE 2
YOU SHOULD SEE THE NUMBER RANGES OF SIGNED AND UNSIGNED NUMBERS:
  SIGNED: -8000000000000000 7FFFFFFFFFFFFFFF
UNSIGNED: 0 FFFFFFFFFFFFFFFF
```

But is it Forth? It passes the Forth-94 Core Test Yes!

```
include core.fs TESTING: CORE WORDS
...
YOU SHOULD SEE THE NUMBER RANGES OF SIGNED AND UNSIGNED NUMBERS:
  SIGNED: -8000000000000000 7FFFFFFFFFFFFFFF
UNSIGNED: 0 FFFFFFFFFFFFFFFF
TESTING: INPUT: ACCEPT

PLEASE TYPE UP TO 80 CHARACTERS:
it works

RECEIVED: "it works"
TESTING: DICTIONARY SEARCH RULES
GDX exists  ok 🙂
```

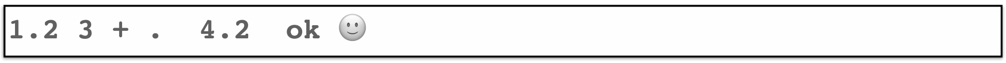
Blending Forth

- But the stack can hold not just (our) numbers.
- It’s implemented as a Python list that can hold any Python object
    - float numbers
    - strings
    - lists and dictionaries
    - method and functions
    - …

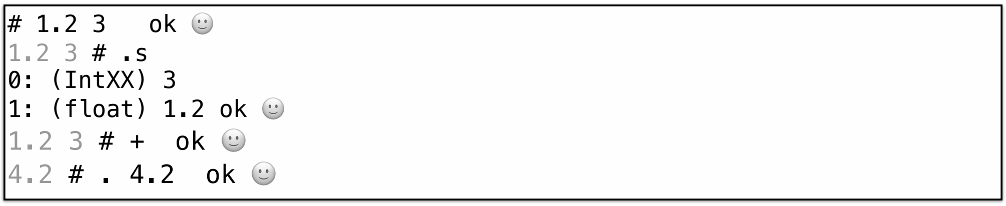
pluggable number system

Blending Forth

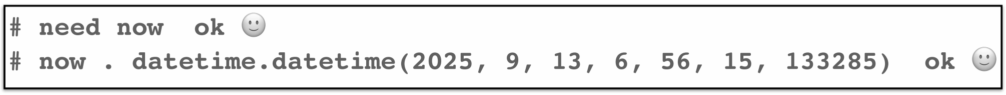
Python Objects on the Data Stack



```
1.2 3 + .  4.2  ok 🙂
```



```
# 1.2 3   ok 🙂
1.2 3 # .s
0: (IntXX) 3
1: (float) 1.2 ok 🙂
1.2 3 # +  ok 🙂
4.2 # . 4.2  ok 🙂
```



```
# need now  ok 🙂
# now . datetime.datetime(2025, 9, 13, 6, 56, 15, 133285)  ok 🙂
```

Related Work

- *oforth* by Franck Bensusan
  - objects on the stack
  - no standard forth syntax (control structures)
  - similar enough to be called Forth

But is it Forth?

"If it walks like a duck and

it quacks like a duck,

then it must be a duck“

Does it?

Demo

Blending Forth

- Why not use Python in first place?
- Forth is concatenative and allows to execute programs interactively step by step.

Blending Forth

Conclusion

• implementing Forth in other languages	Forth inherits their properties
• abstraction and representation	the heart of implementation
• blending Forth	Where to go from here?

Discussion

What is a character?
UTF-8, Unicode, and the Xchar wordset

M. Anton Ertl, TU Wien

Concepts

Concept	UTF-8	Forth
code unit	1 byte	char
code point	1-4 bytes	xchar or string
glyph/character	≥ 1 code point	string

Use Strings!

Usage inside Gforth (2021)

xchar		string	
3	x-size		
1	xc!+	43	move
3	xc!+?	43	move
5	xc,	9	mem,
5	xc-size	114	nip
5	xc@+	43	move
4	xchar+		
1	xemit	151	type
2	xkey		
1	xkey?		

xchar ext		string	
2	+x/string	71	/string
0	-trailing-garbage		
1	ekey>xchar		
6	x-width		
3	xc-width	6	x-width
6	xchar-		
0	xhold	4	holds
1	x\string-	71	/string