Writing a fast theorem prover in Haskell

Nicholas Smallbone
nicsma@chalmers.se
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden

Abstract

Automated theorem provers (ATPs) are natural to implement in a functional language, because they do a lot of symbolic manipulation of terms. Unfortunately, making a fast ATP needs careful control over low-level details of data representation, so the best ATPs today are usually written in C or C++. The author's ATP Twee goes against the grain by being written in Haskell, and yet has state-of-the-art performance. We describe the ideas we use to get good low-level performance even from high-level Haskell code; the most important is using GHC's *pattern synonyms* to separate the high-level view of a datatype from its low-level representation.

ACM Reference Format:
Nicholas Smallbone. 2026. Writing a fast theorem prover in Haskell. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Gothenburg

1 Introduction

Twee [8] is an *equational theorem prover*: a tool that automatically performs equational reasoning. Given a set of equations with free variables (the *axioms*), and a *goal*, it searches for a proof of the goal from the axioms. For example, suppose that we want to prove that, in a group, if $a + b = a$ for some a and b then $b = 0$. We can give Twee the following axioms (where x, y and z are variables, and a and b represent unknown but fixed constants):

```
Axiom 1 (associativity): x + (y + z) = (x + y) + z.
Axiom 2 (plus_zero_left): 0 + x = x.
Axiom 3 (plus_zero_right): x + 0 = x.
Axiom 4 (minus_left): -x + x = 0.
Axiom 5 (minus_right): x + -x = 0.
Axiom 6 (assumption): a + b = a.
```

and ask it to prove the goal $b = 0$. Twee immediately reports the following proof:

```
Goal 1 (goal): b = 0.
Proof:
  b
= { by axiom 2 (plus_zero_left) R->L }
  0 + b
= { by axiom 4 (minus_left) R->L }
  (-a + a) + b
= { by axiom 1 (associativity) R->L }
  -a + (a + b)
= { by axiom 6 (assumption) }
  -a + a
= { by axiom 4 (minus_left) }
  0
```

Twee can be used to reason about a variety of domains (e.g. abstract algebra, program calculation) and puzzles (e.g. showing that the sentence *fruit flies like a banana* can be parsed in two different ways). It can prove difficult problems and is one of the strongest equational theorem provers available today.

Functional languages are a good fit for writing theorem provers, because they are so good at symbolic manipulation. Indeed Twee is written in Haskell, and thanks to Haskell's high-level nature is roughly a tenth the size of its competitors:[1]

Prover	Language	Lines of code (approx.)
Vampire[1]	C++	100000
E[7]	C	100000
Waldmeister[4]	C	65000
Twee	Haskell	7500

Despite this, theorem provers are typically written in system languages such as C and C++. The reason is that, to make a fast theorem prover, we need the core of the prover to be fast and memory-efficient, so we need to carefully design the low-level details of the prover's data structures. This makes the obvious choice a low-level language which gives us that control.

So by writing a theorem prover in Haskell, do we not sacrifice performance? In this paper we show that no, Haskell can compete with low-level languages. We do need to design the core data structures carefully, and this means that the lowest-level parts of Twee are written in a low-level and unpleasant style, but thanks to Haskell's powerful abstraction facilities we can hide these details away, and write everything else in a normal, high-level functional style. The killer feature is GHC's pattern synonyms, which allow us to separate the semantic view of a datatype from how it is represented in memory. Using them, we can design high-level datatypes with low-level representations.

We will focus mainly on *terms*, the core datatype in Twee, and how to use GHC Haskell to present a high-level view of a low-level implementation.

2 Pretty but slow terms

Let us first see how to make a slow theorem prover in Haskell. The core datatype in Twee is the term, which is either a function application or a variable. This is easy to code in Haskell:

[1] Note that Vampire and E are able to reason about formulas in first-order logic and not just equations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, Washington, DC, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
data Term
  = App {-# UNPACK #-} !Fun [Term]
  | Var {-# UNPACK #-} !Var
newtype Fun = Fun Int
newtype Var = Var Int
```

For simplicity we model function and variable symbols as integers, and for efficiency we use an `UNPACK` pragma to store the symbol inline inside the `App` or `Var` constructor.

As a running example of an operation on terms we shall use *matching*. Given a term t and a term *pat*, matching tells us whether *t* is an instance of *pat*, and if it is returns a substitution:

```
match :: Term -> Term -> Maybe Subst
type Subst = Map Var Term
```

For example, if `pat = App f [Var x, Var y]`, and `t = App f [App g [Var x, Var y], App a []]`, then `match pat t` should return the substitution `{x → App g [Var x, Var y], y → App a []}`. Matching is straightforward to implement in Haskell by threading the substitution through:

```
match = term Map.empty
  where
    term :: Subst -> Term -> Term -> Maybe Subst
    term sub (Var x) t =
      case Map.lookup x sub of
        Nothing -> Just (Map.insert x t sub)
        Just u -> if t == u then Just sub else Nothing
    term sub (App f pats) (App g ts)
      | f == g = list sub (unpack pats) (unpack ts)
      | otherwise = Nothing

    list :: Subst -> [Term] -> [Term] -> Maybe Subst
    list sub [] [] = Just sub
    list sub (pat:pats) (t:ts) = do
      sub' <- term sub pat t
      list sub' pats ts
```

This representation of terms is natural, and easy to program with. Perfect; or is it?

Unfortunately, this approach is *far too inefficient* for our needs. A complex proof search is going to do many millions or even billions of term operations, so terms must be stored in a machine-friendly way. Our term datatype is not machine-friendly because it has *too many pointers*. Figure 1 shows how the term `App f [Var x, App g [Var x, Var y]]` looks in memory: the nodes represent heap objects and the edges pointers. Notice that each function argument uses two heap objects, one for the cons cell and one for the argument, and that going from a function to its argument requires traversing multiple pointers. This leads to multiple problems:

(1) Algorithms over these terms will do a lot of pointer-chasing.
(2) Since the data is tree-structured, most algorithms are going to be recursive rather than iterative. For example, in `match`, there is a non-tail recursion from list to term, and also the use of `t == u` is going to recurse over the term structure.
(3) The huge number of objects puts massive pressure on the garbage collector.

In short, we want to eliminate pointers and (non-tail) recursion as far as possible.

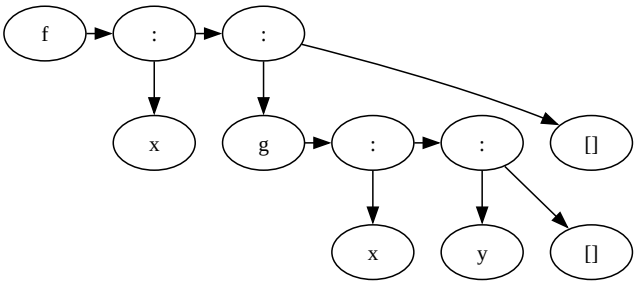


Figure 1: Memory layout of the term f(x, g(x, y))

0	1	2	3	4
f:5	x:1	g:3	x:1	y:1

Figure 2: Stringterm representation of f(x, g(x, y))

3 Ugly but fast terms

Now let us see a machine-friendly term representation. A stringterm [2] represents a term as an array of symbols, shown in Figure 2. This figure represents the term $f(x, g(x)y))$. The array contains the symbols of the term in pre-order traversal order. For each symbol, it also store the length (in symbols) of the subterm rooted at that symbol. For example, in Figure 2, $g : 3$ means that the subterm $g(x, y)$ has a length of 3. From the array and the length information, we can reconstruct the original term.

Recalling that function and variable symbols are integers, we can code each entry of the array as a 64-bit integer, for example using the following scheme:

- 32 bits to store the symbol number (`int symbol`)
- 1 bit to store whether it is a function symbol or variable symbol (`bool isFun`)
- 31 bits to store the length of the subterm rooted at that symbol (`int size`)

This data structure has excellent performance characteristics. First, there are no pointers, so each term is just one object for the garbage collector. Second, since terms are just data, we can compare terms for equality with a single `memcmp`. Third, many term operations naturally become loops. For example, to implement `match pat t`, we can loop through the symbols of `pat`, comparing each one against the corresponding symbol in `t`. Here is how that would look in an imperative pseudocode:

```
match(pat, t) =
  sub := empty substitution
  i := 0; j := 0
  while i < len(pat) and j < len(t):
    if pat[i].isFun:
      if pat[i] != t[j]: fail
      i++; j++
    else:
      x = pat[i].symbol
      size = t[j].size
      if x is bound in sub:
        if sub[x] != t[j..j+size): fail
```

```
      else:
        sub[x] = t[j..j+size)
      i++; j += len
  return sub
```

This code is fast, but it is so low-level! We can imagine transliterating it into Haskell, a tail-recursive function which maintains `i` and `j` as parameters, but why would we? If we are going to write code like this we can just as well do it in C++. What we want is to write the matching code in a high-level, functional style, but have it compile to something like the code above. We will now show how to do this.

4 Pretty and fast terms

We are going to take our stringterm representation and turn it into an idiomatic Haskell API. A term is naturally a recursive type but there is no sign of that in stringterms, where instead everything is done by array indexing; let us first fix that by recovering some recursive structure.

The idea is that a *slice* of a stringterm has a natural recursive structure. For example, the subterms of a stringterm are slices of that stringterm, as we see in the code of `match`: `t[j..j+size)`. Also, suppose that we allow a stringterm to represent a *list* of terms (by concatenation). Then removing the first function symbol from a stringterm gives a stringterm of the function's arguments – for Figure 2, `t[1..5)` gives us x followed by $g(x, y)$.

Hence we arrive at the following design. The basic structure is a *term list*, which we represent as a slice into an array of symbols. As Haskell's arrays do not natively support slicing, we store the slice explicitly as a pair of bounds `[start..end)`. A term is represented as a term list, that we happen to know contains exactly one term:

```
data TermList =
  TermList {
    contents :: {-# UNPACK #-} !ByteArray,
    start :: {-# UNPACK #-} !Int,
    end :: {-# UNPACK #-} !Int }
newtype Term =
  Term {
    singleton :: TermList }
```

As described above, the symbols are coded as 64-bit integers. `ByteArray` (despite the confusing name) is GHC's primitive type for unboxed arrays of all types, and GHC supports treating it as an array of `Int64`s. We can read entries of the array using the following ugly code:

```
data Symbol =
  Symbol {
    isFun :: Bool, -- Is it a function?
    symbol :: Int, -- What is the symbol number?
   size :: Int } -- What is the size of the term rooted here?

-- Convert symbols to and from Int64. The encoding:
-- * bits 0-30: size
-- * bit 31: 0 (variable) or 1 (function)
-- * bits 32-63: symbol
{-# INLINE toSymbol #-}
toSymbol :: Int64 -> Symbol
toSymbol n =
  Symbol (testBit n 31)
    (fromIntegral (n `unsafeShiftR` 32))
    (fromIntegral (n .&. 0x7fffffff))

{-# INLINE fromSymbol #-}
fromSymbol :: Symbol -> Int64
fromSymbol = ...

-- Read a symbol from a ByteArray.
indexSymbol :: ByteArray -> Int -> Symbol
indexSymbol arr n = toSymbol (indexByteArray arr n)
```

As we see, when writing low-level Haskell, the code is sometimes ugly (it would be nicer if GHC supported bitdata [3]). We now aim to show that we can build up from these low-level foundations to an idiomatic Haskell API.

We start by building functions to recursively take apart terms and term lists. A term is either a variable or a function application, and a term list is either empty or non-empty, so we ought to be able to define the following deconstructors:

```
unterm :: Term -> Either Var (Fun, TermList)
unlist :: TermList -> Maybe (Term, TermList)
```

In fact, it is not too hard – we read the first symbol to work out the structure of the term, then just manipulate the slice indexes:

```
unterm (Term (TermList contents start end))
  | isFun sym =
    Right (Fun (symbol sym),
           TermList contents (start+1) end)
  | otherwise =
    Left (Var (symbol sym))
  where
    sym = indexSymbol contents start

unlist (TermList contents start end)
  | start == end = Nothing
  | otherwise = Just
      (Term (TermList contents start mid),
       TermList contents mid end)
  where
    sym = indexSymbol contents start
    mid = start + size sym
```

This is in fact a complete API for deconstructing terms, and we can now write functions that operate on terms without needing to worry about array indexes. But it is a bit ugly to have to write `unterm` and `unlist` everywhere.

Here *pattern synonyms* [6] come to the rescue! Pattern synonyms are a feature for defining new patterns for pattern matching, which look like constructors but desugar in custom ways. We declare two patterns `Nil` and `Cons t ts` for term lists; `Nil` matches any term list whose `unlist` is `Nothing`, and `Cons t ts` matches any term list whose `unlist` is `Just (t, ts)`. Similarly, we declare two patterns `Var` and `App` for terms:

```
pattern Nil :: TermList
pattern Nil <- (unlist -> Nothing)
pattern Cons :: Term -> TermList -> TermList
pattern Cons t ts <- (unlist -> Just (t, ts))
```

```
{-# COMPLETE Nil, Cons #-}

pattern Var :: Var -> Term
pattern Var x <- (unterm -> Left x)
pattern App :: Fun -> TermList -> Term
pattern App f ts <- (unterm -> Right (f, ts))
{-# COMPLETE Var, App #-}
```

(The `COMPLETE` declarations are a hint to GHC's warning machinery that covering those two cases constitutes a complete pattern match.)

Now we can deconstruct stringterms just as if they were members of an algebraic data type! Here, for example, is a function `showTerm` that shows a term as a string:

```
showTerm :: Term -> String
showTerm (Var x) = show x
showTerm (App f Nil) = show f
showTerm (App f ts) =
  show f ++ "(" ++
    intercalate ", " (map showTerm (unpack ts)) ++ ")"

unpack :: TermList -> [Term]
unpack Nil = []
unpack (Cons t ts) = t:unpack ts
```

This is entirely normal-looking Haskell code. The only sign of the underlying stringterm representation is that `ts` is a `TermList`, and must be explicitly converted to `[Term]` if we want to apply list functions such as `map` on it. And it compiles to efficient code – each pattern match desugars to a call to `unterm` or `unlist`, which reads one symbol from the array and does a small amount of integer arithmetic on the slice indexes.

4.1 A souped-up term list API

The pattern synonyms above provide efficient access to the underlying stringterm, but they expose it only as a recursive structure. We saw in Section 3 that matching is more efficiently implemented as a loop, which in a functional setting corresponds to tail recursion. We shall now extend our API to support this.

Given a non-empty term list, we already know that two components can be extracted from it:

- The *head*, the first term in the list
- The *tail*, everything but the first term in the list

But we can also define a third component:

- The *rest*, everything but the first *symbol* in the list

In Figure 2, the rest consists of the slice starting at position 1, containing the terms x and $g(x, y)$. Semantically, `rest (App f ts:us) = ts ++ us`. Using the rest allows us to iterate through a term symbol by symbol.

We therefore add one more pattern synonym: `ConsSym{hd, tl, rest}`. This matches a term list whose head is `hd`, tail is `tl` and rest is `rest`.

Using this pattern, we can finally define an efficient and functional implementation of `match`. It uses a helper function `matchListIn` which (1) works on term lists, and (2) takes a substitution to be threaded through:

```
match :: Term -> Term -> Maybe Subst
match pat t =
  matchListIn Map.empty (singleton t) (singleton u)

matchListIn :: Subst -> TermList -> TermList -> Maybe Subst
matchListIn !sub Nil Nil = Just sub
matchListIn sub
  ConsSym{hd = pat, tl = pats, rest = pats'}
  ConsSym{hd = t, tl = ts, rest = ts'} =
    case (pat, t) of
      (Var x, _) ->
        case Map.lookup x sub of
          Nothing ->
            let sub' = Map.insert x t sub in
            matchListIn sub pats ts
          Just u | t == u ->
            matchListIn sub pats ts
      (App f _, App g _)
        | f == g -> matchListIn sub pats' ts'
        | otherwise -> Nothing
```

Note that the function is tail-recursive; the trick we use to get this is that, if `pat` and `t` start with the same function symbol, we recurse into the *rest* rather than the tail (`matchListIn sub pats' ts'`). GHC transforms it into an efficient tail-recursive loop that matches the code in Section 3, and compiles to efficient and direct machine code:

- The worker-wrapper transform unboxes the term lists, so both `pat` and `t` become three arguments: a pointer to the `contents` array, and the values of `start` and `end` as unboxed `Int64`s.
- Matching on `Nil` desugars into checking whether `start == end`.
- Matching on `Var` and `App` desugars into reading the symbol from the array and doing bit operations to decode it.
- The call to `t == u` turns into a `memcmp`.
- The recursive calls to matchListIn turn into a tail recursion that just advances the `start` of `pat` and `t`.

The implementation of `match` seen above is a little stylised, for example carefully doing no more pattern matching than necessary, but we are willing to accept this as it is one of the most performance-sensitive functions in Twee. For less performance-sensitive functions (such as `showTerm` above, and in fact most code in Twee), we can write entirely unremarkable Haskell code.

5 Building terms, take 1

What is wrong with the following code for substitution?

```
subst :: Subst -> Term -> Term
subst sub (Var x) =
  Map.findWithDefault (Var x) x sub
subst sub (App f ts) =
  App f (substList ts)
    where
      substList Nil = Nil
    substList (Cons t ts) = Cons (subst sub t) (substList ts)
```

Answer: it does not compile! The patterns `Var`, `App`, `Cons` and `Nil` are *unidirectional* pattern synonyms: they can be used in patterns but not to construct terms. We could make them bidirectional, but

that is inappropriate: a constructor should take *constant time*, but creating (e.g.) the term `App f ts` out of `f` and `ts` takes linear time: first a new array needs to be created to hold the term, then `f` needs to be written there, then the entire contents of `ts` needs to be copied there.

We solve this problem using the well-known *builder* approach. The idea is to define a type `Builder` which represents a function that builds a term list. A builder for the term `t` is a function that takes an array and writes `t` there. We define a typeclass for things that have builders:

```
class Build a where
  builder :: a -> Builder
instance Build Builder where ...
instance Build Term where ...
instance Build TermList where ...
instance Build Var where ...
instance Build a => Build [a] where ...
```

Then we create a builder API mirroring the term API:

```
var :: Var -> Builder
app :: Build a => Fun -> a -> Builder
```

(We declare `Builder` as a monoid that concatenates terms, so there is no need for an explicit `nil` and `cons`.) Finally we define functions for executing the builder:

```
build :: Build a => a -> Term
buildList :: Build a => a -> TermList
```

With the builder API in hand, we can define substitution as follows:

```
subst :: Subst -> Term -> Term
subst sub t = build (substBuilder sub t)

substBuilder :: Subst -> Term -> Builder
substBuild sub (Var x) =
  case Map.lookup x sub of
    Nothing -> var x
    Just t -> builder t
subst sub (App f ts) =
  app f (substList ts)
    where
      substList Nil = mempty
      substList (Cons t ts) =
        substBuilder sub t <> substList ts
```

All functions in Twee that create terms have to be defined in this builder style, which is undeniably clunky. To solve this, we have recently started experimenting with a more natural way of creating terms – with help from our old friend pattern synonyms.

6 Building terms, take 2

Let us assume that the stringterm API is available in the module `String`. The idea is to define a term type which represents a term as either a stringterm or a more traditional algebraic data type (let us call that a *nested* term):

```
data Term =
    StringTerm {-# UNPACK #-} !String.Term
  | VarTerm {-# UNPACK #-} !Var
  | AppTerm {-# UNPACK #-} !Fun ![Term]
```

Notice that we can freely mix the two types of terms, e.g. `AppTerm f [StringTerm t, Var x]`.

Now, these terms are easy to construct, and can be pattern matched as usual. But each kind of term can be represented in two different ways, either as a stringterm or a nested term. For example, the variable `x` could be represented as both `VarTerm x` and `StringTerm (String.Var x)`.

So we introduce *pattern synonyms* that abstract over the different ways of representing each term. We define a deconstructor function for terms:

```
unterm :: Term -> Either Var (Fun, [Term])
unterm (StringTerm (String.Var x)) = Left x
unterm (StringTerm (String.App f ts)) =
  Right (f, String.unpack ts)
unterm (VarTerm x) = Left x
unterm (AppTerm f ts) = Right (f, ts)
```

Now we can define pattern synonyms just the same as for our stringterm API. The only difference is that these synonyms are *bidirectional*, so can also be used to construct terms:

```
pattern Var :: Var -> Term
pattern Var x <- (unterm -> Left x) where
  Var x = VarTerm x
pattern App :: Fun -> [Term] -> Term
pattern App f ts <- (unterm -> Right (f, ts)) where
  App f ts = AppTerm f ts
{-# COMPLETE Var, App #-}
```

Now all our term-manipulating functions can be defined by normal pattern matching. For example, here is substitution:

```
subst :: Subst -> Term -> Term
subst sub (Var x) = Map.findWithDefault (Var x) x sub
subst sub (App f ts) = App f (map (subst sub) ts)
```

Notice that `Var` and `App`, when used as constructors, return nested terms. So, if we transform a stringterm, parts of it will become nested, and as we transform it more and more, operations on it will become less machine-friendly. We therefore add an operation that transforms a term to a stringterm using the builder API:

```
toStringTerm :: Term -> String.Term
```

and using that we can define a function of type `Term -> Term` that always returns a stringterm:

```
flatten :: Term -> Term
flatten t = StringTerm (toStringTerm t)
```

Then, in strategically chosen spots in Twee, we can flatten important terms, making sure that future operations on them are more efficient. Note too that the constructor `StringTerm` is exported, so functions that need extra efficiency (such as `match`) can be special-cased for stringterms.

We currently have an implementation of this API which seems to generate efficient code, but have not yet integrated it into Twee to test its real-world performance.

7 The rest of the core

Talk about other places we are careful about data representation:

- Interning
- Batched heap

- Term indexing

8 Limitations

Talk about:

- Manual worker-wrapper in builder
- Need to cache symbol in Term type to avoid repeated cases in generated code
- Manually monomorphic types in a few places

9 Evaluation

To do. But: switching terms from a datatype to a pattern synonym over a stringterm improves prover performance by about 3x.

10 Conclusion

It is well known that "Haskell is the world's finest imperative programming language" [5]; we believe that Haskell is also a fine low-level language. Writing the low-level parts is not as nice as in (say) C, but thanks to Haskell's abstraction powers and especially pattern synonyms, we can build a high-level Haskell program that runs on top of a low-level Haskell core.

References

[1] Filip Bártek, Ahmed Bhayat, Robin Coutelier, Márton Hajdú, Matthias Hetzenberger, Petra Hozzová, Laura Kovács, Jakob Rath, Michael Rawson, Giles Reger, et al. 2025. The vampire diary. In *International Conference on Computer Aided Verification*. Springer, 57–71.

[2] Jim Christian. 1993. Flatterms, discrimination nets, and fast term rewriting. *Journal of automated reasoning* 10, 1 (1993), 95–113.

[3] Iavor S Diatchki, Mark P Jones, and Rebekah Leslie. 2005. High-level views on low-level representations. *ACM SIGPLAN Notices* 40, 9 (2005), 168–179.

[4] Bernd Löchner and Thomas Hillenbrand. 2002. A Phytography of WALDMEISTER. *AI Communications* 15, 2,3 (Aug. 2002), 127–133.

[5] Simon Peyton Jones. 2001. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*. IOS Press, 47–96. https://www.microsoft.com/en-us/research/publication/tackling-awkward-squad-monadic-inputoutput-concurrency-exceptions-foreign-language-calls-haskell/

[6] Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A Eisenberg. 2016. Pattern synonyms. In *Proceedings of the 9th International Symposium on Haskell*. 80–91.

[7] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. 2019. Faster, Higher, Stronger: E 2.3. In *Automated Deduction – CADE 27*, Pascal Fontaine (Ed.). Springer International Publishing, Cham, 495–507. https://doi.org/10.1007/978-3-030-29436-6_29

[8] Nicholas Smallbone. 2021. Twee: An Equational Theorem Prover.. In *CADE*. 602–613.