Termination Checking: Comparing Structural Recursion and Sized Types by Examples

David Thibodeau

Decemer 3, 2011

Abstract

Termination is an important property for programs and is necessary for formal proofs to make sense. In order to make sure that a program using recursion is terminating, one can use a termination checker that will use some strategy to verify the termination of a given program. Two very common strategies are structural recursion that makes an analysis on the structure of terms and sized types that use size annotation in the types. In this paper, we compare the two approaches to identify their strengths and limitations and to see how one can proceed to prove termination using each of the strategies. This analysis is done on different examples: a series of functions on the natural numbers and a mergesort algorithm. We also discuss the use of each of the approaches in the cases of proofs.

1 Introduction

In order to ensure correctness, one needs to check that termination is achieved. One major, but unfortunate, result of computability theory is that the halting problem is undecidable. To enforce it, one can restrict its semantics to a normalizing calculus so that no non-terminating function can be written. As this approach prohibits the use of recursion, this limits greatly what programs can be implemented. For most purposes, this is inadequate. Instead, we allow recursion and pass a termination checker on the program. Such termination checker will make sure that the recursive calls are made on smaller arguments with respect to a wellfounded order. There are two main approaches that are used to implement termination checkers. These two approaches are structural recursion, which is used by systems such as Twelf [PS02] and Agda [AA02]; and sized types, used in MiniAgda [Abe10] and ATS [Xi04].

Structural recursion defines an order on the structure of terms. A term M is usually said to be smaller than a term N if we obtain N by applying one or more constructors C on M. The ordering appears directly by the use of pattern matching on the argument. Termination checking on the functions can be done without having to annotate function with extra information. The ordering is infered from the terms themselves and wellfoundedness is infered from the constructors.

With sized types, as it is presented in [Abe10], types are annotated with a size argument that determines an upper bound for term. The size annotation of a term represents the maximal height of a tree with nodes as constructors of the term. The size argument is a natural number or an ordinal since the height of the tree can be transfinite. The use of subtyping allows the use of different upper bounds for a term. Then, there is quantification over size variable in the type of a function definition and size arguments are passed through recursive calls, making sure each time that they get smaller. It is also possible to define size variables satisfying constraints such as defining a variable j such that $j < i$ for some predefined i. Then termination is ensured by well-foundedness of the natural numbers.

In this paper, we will study the use of structural recursion and sized types for termination checking. We wish to make a comparison between the two approaches to see how they work, what are their strengths and what are their limitations.

A priori, the biggest advantage of structural recursion is that the process is done in background: assuming that the termination checker can find a suitable ordering on terms, there should be little overhead for the user, thus simplifying the code. Programmers would then be able to use such checker without having to modify their original code. However, this approach has its limitations. The termination checker could, in principle, be not able to assert the termination of a function because it is not obvious for the checker that the recursive calls are made on arguments that are strictly smaller. We also wish to analyse when this does happen. On the other side, it seems that sized types would shine by their flexibility and power: since more information is tracked through the program, it would probably be easier for sized types to identify that a function is indeed terminating. However, we would have to pay some cost, namely, having to carry around size annotations. We wish to verify if these preconceptions hold and, if so, to what extend.

To do this, we will go through some examples of functions and test how do we achieve to obtain, for each approach, an implementation that is accepted. To study structural recursion, we will use the functional programming language Agda that uses the termination checker Foetus [AA02]. To study sized types, we will use MiniAgda [Abe10]. In section 2, we will have a look at different functions on the natural numbers such as the minus function, the division function and the Euclidean algorithm. In section 3, we will go through the merge sort algorithm. In section 4, we will discuss of the use of termination checkers for proofs.

2 Functions on Natural Numbers

We first need to define our natural numbers within Agda and MiniAgda respectively. The definition is inductive, using zero and a successor function. In Agda, the representation is the following.

```
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat
```

In MiniAgda, we define the natural numbers with size annotation

```
sized data SNat : Size -> Set
{ zero : [i : Size] -> SNat ($ i)
; succ : [i : Size] -> SNat i -> SNat ($ i)
}
```

We note that in MiniAgda, size arguments must be provided explicitely to the constructors. The reason is that MiniAgda is an experimental system whose goal is to investigate sized types which does not have reconstruction. According to [Abe10], in a mature system such as Agda, sized annotation would be reconstructed during runtime. This point is important for our analysis: size annotation will make the code more complicated and so harder to read, but size reconstruction would account for that problem. This raises several questions such as can all size annotations be reconstructed or would some size annotations still be required, and how would that affect the classe of programs that can be certified to be terminating.

We first present a minus function in Agda

```
minus  : Nat -> Nat -> Nat
minus zero y = zero
minus x zero = x
minus (succ x) (succ y) = minus x y
```

and in MiniAgda

```
fun minus : [i : Size] -> SNat i -> SNat # -> SNat i
{   minus i (zero (i > j)) y = zero j
;   minus i (succ (i > j) x) (zero .#) = succ j x
;   minus i (succ (i > j) x)   (succ .# y) = minus j x y
}
```

Both of these implementation are accepted by the termination checker. By striping off the constructor succ from each of the two arguments, we make a recursive call on an argument that is structurally smaller. It is also captured by the size argument. Since the size annotation represents the height of the derivation of our natural number, when we extract x from the equation $succ\ x$, we reduce the height of the derivation by one.

Then, we present a division function for MiniAgda.

```
fun div : [i : Size] -> SNat i -> SNat # -> SNat i
{ div i (zero (i > j))          y = zero j
; div i (succ (i > j) x) y = succ j (div j (minus j x y) y)
}
```

Here, we call recursively the division function with argument $x - y$ instead of $(succ\ x) - (succ\ y)$ Since the output of the minus function is of size as big as the first argument, the output will be smaller than the input argument *succ x*. Thus, the recursive call is made on smaller arguments. One cannot use the same idea with structural recursion since the termination checker has no way to identify if the result of *minus x y* is indeed smaller than *succ x*.

In order to write a division function that is accepted by the termination checker, one needs to do more. The strategy that we present here is described in [BD09]. We will add a third argument to the division function. This argument must be a proof term of a domain predicate that expresses that for a given pair of natural numbers the division terminates. We define this predicate in the following way.

```
data DivDom : Nat -> Nat -> Set where
  div-dom-lt : (m n : Nat) -> n > m  -> DivDom m n
  div-dom-geq : (m n : Nat) -> n ≤ m -> DivDom (m - n) n ->
                DivDom m n
```

The division function becomes

```
div : (m n : Nat) -> DivDom m n -> Nat
div .m .n (div-dom-lt m n p) = zero
div .m .n (div-dom-geq m n p q) = (succ zero) + (div (m - n) n q)
```

and is accepted by the termination checker since the proof term gets structurally smaller at each recursive call.

While this provides a way to prove termination with structural recursion, in its current form it requires us to provide a proof term each time we want to use the function. This is unpractical. To not have to do that, we need to prove that for all m and all $n > 0$, there is a *DivDom m n*. Then, instead of giving a proof term, we invoke this theorem. Proving that *DivDom* holds for all natural numbers is not trivial. In addition, we have to consider how this approach scales with the complexity of the problem. For this example, the predicate was quite simple, it is believable that as the algorithms grow in complexity, the predicate will also grow in complexity since it must account for all the cases. In comparison, sized types did not need to provide such predicate and prove that they hold for all input and so appears to be more practical.

We wish to see if these observations hold for a more complicated example. We now have a look to the Euclidean algorithm.

2.1 Greatest Common Divisor

The Euclidean algorithm is the standard algorithm to find the greatest common divisor of two numbers. To get $\gcd(a, b)$, we first assume that $a \geq b$. If it is not the case, we can always swap a and b. We compute the remainder of the division of a by b and write $a = x_0 b + r_0$, where x_0 is the quotient and r_0 is the remainder. Then, we compute $\gcd(b, r_0)$, that is, we write $b = x_1 r_0 + r_1$. We continue this process until we get $r_{n-1} = x_{n+1}r_n + r_{n+1}$ and $r_n = x_{n+2}r_{n+1}$ so that r_{n+1} divides exactly r_n. Then, since r_{n+1} divides both r_{n+1} and r_n, it must also divide r_{n-1}. And it also divide r_i for all $0 \leq i \leq n+1$ so that it must also divide b and a. By construction, it is the greatest common divisor.

To implement it in Agda, one might want to proceed in the following fashion.

```
gcd : Nat -> Nat -> Nat
gcd zero y = y
gcd x zero = x
gcd x y = if (y ≤ x)
      then (gcd (x - y) y)
      else (gcd y x)
```

The two first cases handle special cases. The first one will actually be used at the last recursive call. Let us analyse what this code does. Supposing that x and y are not zero, we verify if y is smaller than x. It is the case, we call gcd again with argument $x - y$ and y. Instead of using a division with remainder, we substract successively y from x until we obtain an x' strictly smaller than y. When x' is strictly smaller than y, it corresponds to the remainder of the division of x by y. At this point, we compute the greatest common divisor of y and x'. If we get to the point that $x' = y$, $x' - y = 0$ is sent as first argument of the function, the function will return y.

This function is terminating. If we want to prove that informally, we observe that if $y \leq x$ and $y > 0$, the first argument will pass from x to $x - y$ and so will get smaller and the second argument, y, will stay constant. But if $y > x$, then the second argument passes from y to x and so gets smaller by assumption and the first argument grows because it passes from x to y. Therefore, either the second argument decreases or it stays constant while the first argument decreases. Thus, the function is recursively called on smaller arguments with respect to a lexicographic order bases on the second argument.

However, while we have that the arguments get smaller, they do not get structurally smaller as $x - y$ and y are not structurally smaller than x. Thus, Agda's termination checker cannot assert that the function is terminating.

To obtain termination, we can use a domain predicate for the algorithm as we did for the division function. However, there exists another similar method that one can use to obtain termination with structural recursion. It consists of writing an accessibility predicate on a binary relation and defining a set of elements that have no infinite descending sequences with respect to this relation. We say that these elements are accessible. One can find a more detailed treatment of the accessibility predicate in [Nor88]. In Agda, we can define the accessibility predicate as follow.

```
data acc {A} (gt : A -> A -> Set) (n : A) : Set where
 nacc : (∀ {m} -> gt n m -> acc gt m) -> acc gt n
```

To prove the termination of our algorithm, we append a proof term that the input is accessible. In spirit, this method is very similar to the domain predicate method: we create a proof term and this proof term will get structurally smaller at each recursive call. In our case, the binary relation is the relation $>$ (greater than) for natural numbers. We note again that in order to not have to construct each time a proof that each of the input are accessible, we must prove that for all natural number n, there is a term $acc \_ > \_ n$. This proof can easily be done by strong induction. Then, we can use it to obtain the following code.

```
gcd : (n : Nat) -> acc _>_ n -> (m : Nat) -> acc _>_ m -> Nat
gcd x p y p2 with islessthan? y x
gcd x (nacc y0) zero p2 | less y' = x
gcd x (nacc y0) (succ y) p2 | less y' =
      gcd (x - (succ y)) (y0 (lem4 y') ) (succ y) p2
gcd zero p y (nacc y1) | greater y' = y
gcd (succ x) (nacc p) y (nacc y1) | greater y0 =
      gcd y (nacc y1) (succ x) (y1 y0)
```

This code is accepted by the termination checker. It has a lot of overhead; we pass two additional proofs of accessibility of n and m. Termination is ensured using these arguments. We also need to prove several lemmas including that if $x > y$ then $x > (x - \text{succ } y)$, which make the algorithm more difficult to implement. This approach has the advantage that we can reuse the accessibility predicate for other functions with natural numbers such that the division function. However, it requires to prove that the argument that you use is indeed smaller than the one before.

Let us have a look to the Euclidean algorithm in MiniAgda. We can write the following algorithm.

```
fun gcd : [i : Size] -> [j : Size] -> SNat i ->
          SNat j -> SNat (max i j)
{ gcd i j (zero (i > k))   y                = y
; gcd i j (succ (i > k) x) (zero (j > l))   = succ k x
; gcd i j (succ (i > k) x) (succ (j > l) y) =
    branch (max i j) (isSmaller (max i j) x y)
           (gcd i l (succ k x) (minus l y x))
           (gcd k j (minus k x y) (succ l y))
}
```

This algorithm is very similar to what we have presented at first. There is one distinction. We made the two branches somewhat symmetric and used the minus function on the two arguments with their successor constructed striped. Since at each recursive calls, one of the two arguments gets smaller.

It is important to note that we could not have swapped the two arguments as we cannot verify that the size of one is indeed smaller than the other. To our knowledge, there is no way to obtain a proof that $i < j$ given that $x < y$ for x and y, natural numbers of size i and j respectively. The reason for this is that in MiniAgda, sizes live on another level. While this did not appear to cause problem here, one might find this limitating. However, this is specific of MiniAgda: ATS [Xi04] asks that one provides a metric to every function to indicate what is the argument that gets smaller. In the case of natural numbers, one would use the system defined natural numbers and would provide as a metric one or both of the arguments. Then, by doing the computation that $x < y$, we obtain that the size is indeed smaller since there is no distinction between the two. Also, one would be able to know that $x - y < x$ for $x, y > 0$ since such constraint will be solved using the system's constraint solver. More details of this approach can be found in [Xi01].

With these examples in mind, it appears that sized types capture more naturally the termination of algorithms dealing with natural numbers. In the next section, we study how termination checking scales for lists and for the merge sort algorithm.

3 Merge Sort

Let us now look at the merge sort algorithm on lists of natural numbers. Merge sort works in two parts, it first splits recursively the list in two lists of about equal length, then merges the two lists into one ordered list. Our lists are defined in Agda in the following way:

```
data List : Set where
  [] : List
  _::_ : Nat -> List -> List
```

While we define them in the following way for MiniAgda

```
sized data List (Nat : Set) : Size -> Set
{ nil : [i : Size] -> List Nat $i
; cons : [i : Size] -> Nat -> List Nat i -> List Nat $i
}
```

We implement the mergesort algorithm in Agda as presented in figure 1.

Let us look at what this code does. The merge function calls itself recursively, but at each call, one of the argument gets smaller since we remove a cons operator. it is accepted by the termination checker.

For the mergesort function, we use a split function to split the list in two parts of about the same size. Then, we call mergesort on each of these two lists. The function terminates since it is applied to arguments that are by construction smaller than the original ones. However, this assumes that the split function outputs a pair of lists of size smaller than the original one. Since the original list has at least 2 elements, we know it is the case. However, the termination checker has no information about the length of the output split and so cannot assert termination. To convince the termination checker, one would have to provide a proof that split actually reduces the size of the list.

We now observe how one would implement the algorithm in MiniAgda. The implementation appears in figure 2. If we exclude size annotations and specifics constructs such as the case pattern, we observe that the implementation is very

```
merge : List -> List -> List
merge [] ys = ys
merge xs [] = xs
merge (x :: xs) (y :: ys) =
      if x ≤ y
      then (x :: (merge xs (y :: ys))
      else (y :: (merge (x :: xs) ys))

mergesort : List -> List
mergesort [] = []
mergesort (x :: []) = x :: []
mergesort (x :: (y :: xs)) with split (x :: (y :: xs))
mergesort (x :: (y :: xs)) | pair a b =
      merge (mergesort a) (mergesort b)
```

Figure 1: Merge sort algorithm implemented in Agda

```
fun merge : [i : Size] -> List Nat i -> List Nat i -> List Nat #
{ merge i (nil .Nat (i > k)) b = b
; merge i a (nil .Nat (i > k)) = a
; merge i (cons .Nat (i > k) a as) (cons .Nat (i > l) b bs) =
      case (isSmaller a b)
      { true -> cons Nat # a (merge i as (cons Nat l b bs))
      ; fale -> cons Nat # b (merge i (cons Nat k a as) bs)
      }
}
fun mergesort : [i : Size] -> List Nat i -> List Nat #
{ mergesort i (nil .Nat (i > j)) = nil Nat j
; mergesort i (cons .Nat (i > j) x (nil .Nat (j > k))) =
      (cons Nat j x (nil Nat k))
; mergesort i (cons .Nat (i > j) x (cons .Nat (j > k) y zs)) =
      case split k zs
      { (pair .(List Nat k) .(List Nat k) xs ys) ->
        merge # (mergesort j (cons Nat k x xs))
                (mergesort j (cons Nat k y ys))
      }
}
```

Figure 2: Merge sort algorithm implemented in MiniAgda

similar to the one that we presented above for Agda. This code is accepted by MiniAgda's termination checker. The reason for that lies in the type of the split function which is

```
 fun split : [i : Size] -> List Nat i ->
        Pair (List Nat i) (List Nat i)
```

It asserts that the size of each of the lists in the output is at most the size of the input list. Then, when we case analyse on the list zs, which has size k, we obtain two lists that xs and ys that have size at most k. We then call recursively mergesort on the list cons x xs and cons y ys which have both size j and then merge the two lists. Since our original list had size i and $i > j$, our recursive calls are decreasing. Thus, the termination checker accepts the program.

4 Proofs

We now want to talk about using termination checking for proofs instead of programs. For this, we will use a proof that small-step semantics for numbers is deterministic. The implementation of the proof in Agda was written by Prof. Brigitte Pientka. We first define our semantics.

```
data Tm : Set where
  z       : Tm
  succ    : Tm -> Tm
  pred    : Tm -> Tm
```

Since the definition is similar for MiniAgda, we omit it. We note that since we will not be using functions on these terms directly, we do not need to annotate them with sizes. Then, we define our reductions:

```
data Step : Tm -> Tm -> Set where
  s_succ         : {n : Tm} -> {m : Tm} ->
                   Step n m -> Step (succ n) (succ m)
  s_pred_zero    : Step (pred z) z
  s_pred_succ    : {m : Tm} -> Num_Value m ->
                   Step (pred (succ m)) m
  s_pred         : {n : Tm} -> {m : Tm} ->
                   Step n m -> Step (pred n) (pred m)
```

In MiniAgda, we annotate them with sizes:

```
sized data Step : (i : Size) -> (n, m : Tm) -> Set
{ s_succ :      [i : Size] -> [n, m : Tm] ->
               Step i n m -> Step $i (succ n) (succ m)
; s_pred_zero : [i : Size] -> Step $i (pred z) z
; s_pred_succ : [i : Size] -> [n : Tm] ->
               Step $i (pred (succ n)) n
; s_pred :      [i : Size] -> [n, m : Tm] -> Step i n m ->
```

```
                Step $i (pred n) (pred m)
}
```

We note that in the MiniAgda implementation, the rule s_pred_succ does not take a term of type Num_Value n. The reason for it is that in the Agda proof, we let Agda reconstruct that term. Since MiniAgda does not do any reconstruction, we could not write this rule in the proof as we did not find a way to write it explicitely. Since this is a limitation of MiniAgda that is not related to the termination checker, we chose to modify slightly the semantics to be able to write a proof.

Now, let us have a look to the proof per se. We first start with the Agda implementation.

```
det : {m : Tm} -> {n1 : Tm} -> {n2 : Tm} ->
      Step m n1 -> Step m n2 -> Eq n1 n2
det (s_succ t)      (s_succ t')          with det t t'
... | ref = ref
det (s_pred_zero)   (s_pred_zero)      = ref
det (s_pred_succ _) (s_pred_succ _)    = ref
det (s_pred t)      (s_pred t')          with det t t'
... | ref = ref
-- Impossible cases for pred
det (s_pred t)   (s_pred_succ nv) with values_dont_step t (v_s nv)
... | ()
det (s_pred_succ nv)(s_pred t)    with values_dont_step t (v_s nv)
... | ()
det (s_pred ())     (s_pred_zero)
det (s_pred_zero)   (s_pred ())
```

This proof is accepted by the termination checker because at each recursive call, the term is struturally smaller. This kind of proof is done inductively. When these proofs are encoded in a program, a recursive call corresponds to the use of the induction hypothesis. Our terms being defined inductively with constructors, the induction hypothesis is on terms that are structurally smaller by construction. In this setting, structural recursion appears quite naturally.

Let us look at the MiniAgda implementation

```
fun det : [i : Size] -> [n, m1, m2 : Tm] ->
          Step i n m1 -> Step i n m2 -> Eq m1 m2
{ det i .(succ n) .(succ m1) .(succ m2)
      (s_succ (i > k) .n m1 t) (s_succ (i > l) n m2 t')  =
        cong_succ m1 m2 (det (max k l) n m1 m2 t t')
; det i .(pred z) .z .z (s_pred_zero (i > k))
      (s_pred_zero (i > l)) = ref z
; det i .(pred (succ n)) .n .n (s_pred_succ (i > k) n)
      (s_pred_succ (i > l) .n) = ref n
; det i .(pred n) .(pred m1) .(pred m2)
```

```
      (s_pred (i > k) .n m1 t) (s_pred (i > l) n m2 t')  =
        cong_pred m1 m2 (det (max k l) n m1 m2 t t')
```

We left out the impossible cases since, as MiniAgda does not check for totality, there is no construct to indicate the empty pattern. We note that instead of using a case on *det* (max k l) n $m1$ $m2$ t t' then using reflexivity, we call a function cong_succ (or cong_pred). This function is defined in the following way

```
fun cong_succ : [n, m : Tm] -> Eq n m -> Eq (succ n) (succ m)
{ cong_succ .n .n (ref n) = ref (succ n)
}
```

and is used since pattern matching in MiniAgda, unlike Agda's pattern matching, it does not unify $m1$ and $m2$ to allow the use of reflexivity. If we omit these specific distinctions between the two languages that are not related to their termination checkers, the implementation of the proof in MiniAgda has the same structure as the implementation in Agda. The only distinction is the size annotations. While they do make the code more difficult to understand, as we said before, we can assume that these would be hidden in a mature system.

5 Conclusion and Future Work

We presented four examples of programs implemented in Agda and MiniAgda and studied how the termination checker of each of the systems react when trying to assert their termination. These programs were a minus function, a division function, a greatest common divisor algorithm and a merge sort algorithm. We also implemented a proof that small-step semantics for numbers is deterministic and discussed how termination checkers work when dealing with proofs.

While we were able to make MiniAgda's termination checker accept all our programs without too much effort, things were different for Agda's termination checker. Since $x - y$ is not structurally smaller than x, both the division and the greatest common divisor functions were not accepted by it. The merge sort algorithm was also problematic for structural recursion since a split function does not produce lists that are structurally smaller.

However, it was still possible to create implementations that were accepted. It required to create a predicate that stating the wellfoundedness of the ordering relation for the inputs or stating the termination of the algorithm on the input. Then we had to provide to the function an appropriate proof term for this predicate. To remove the necessity of constructing each time a proof term for the input, one would typically want to prove that the predicate hold for the entire domain of the function. This approach can be quite tedious and cause a lot of overhead in the implementation.

When we studied the used of termination checkers on proofs, the situation appeared to be different. In this setting, the use of structural recursion was quite natural since recursive calls correspond to uses of the induction hypothesis. Thus, one needs the induction hypothesis to apply on terms that are structurally smaller than what you are trying to prove. Hence, the use of structural recursion matches directly what you would expect in verifying a proof.

Using sized types did not cause problem either. They are also able to represent proofs without much effort since the sizes represent the height of the derivations. Thus, if the size decreases, the argument is smaller and the induction hypothesis holds.

However, for proofs, it seemed that the size annotations were unnecessary and were just making the code harder to read. We noted that, while in MiniAgda the size annotations are explicit, one would expect that in a mature system, these would be hidden and reconstructed automatically. One can therefore think that the implementation of a proof would be identical for each of the two approaches.

While we have not investigated reconstruction, we believe that there would be a trade of between the power a termination checker that uses sized types and how much of the size annotations can be reconstructed. However, we believe that a completely automatic sized assignment would still allow the termination checker to accept programs that are accepted by a structural recursion analysis. The reason for that belief is that termination via structural recursion is done by an untyped analysis on the terms, this analysis can be done during reconstruction to infer the appropriate sizes for the terms.

We therefore conclude that in addition of being more powerful than structural recursion, sized types are more flexible. We think that sized types should be prefered over structural recursion for termination checking.

We would, however, investigate these two approaches in the case of coinduction. Since coinductive structures are infinite, it might be difficult to use sized types to represent coinductive structures. We would like to see if this would cause problems that do not occur with structural recursion and, if so, if these problems can be solved. This was not done here because this would require to investigate more on coinduction.

If the investigation of the use of sized types for coinduction give positive results, we would like to have a look to reconstruction. We would like to see how it would be done and if there is indeed a trade-off between the power of termination checker and how much of size annotations must be provided.

6 Acknowledgements

We would like to thank Andrew Cave for helping with the Agda code and Prof. Andreas Abel for answering my questions about MiniAgda. Presenting terminating algorithms for both Agda and MiniAgda would not have been possible without them.

References

[AA02] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *J. Funct. Program.*, 12:1–41, January 2002.

[Abe10] Andreas Abel. MiniAgda: Integrating sized and dependent types. In *Workshop on Partiality And Recursion in Interactive Theorem Provers (PAR)*, July 2010.

[BD09] Ana Bove and Peter Dybjer. *Dependent Types at Work*, pages 57–99. Springer-Verlag, Berlin, Heidelberg, 2009.

[Nor88] B. Nordström. Terminating general recursion. *BIT*, 28:605–619, July 1988.

[PS02] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.4 edition, December 2002. Available as Technical Report CMU-CS-98-173.

[Xi01] Hongwei Xi. Dependent types for program termination verification. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 231–, Washington, DC, USA, 2001. IEEE Computer Society.

[Xi04] Hongwei Xi. Applied type system (extended abstract. In *In post-workshop Proceedings of TYPES 2003*, pages 394–408, 2004.