Development and verification of arbitrary-precision integer arithmetic libraries

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580 Sciences et technologies de
l'information et de la communication (STIC)
Spécialité de doctorat: Informatique
Unité de recherche: Université Paris-Saclay, Inria, Inria
Saclay-Île-de-France, 91120, Palaiseau, France
Référent: Faculté des sciences d'Orsay

**Thèse présentée et soutenue à Orsay,
le 3 novembre 2020, par**

Raphaël RIEU

Composition du jury :

Catherine Dubois Professeure, ENSIIE	Présidente
Karthikeyan Bhargavan Directeur de recherche, Inria	Rapporteur & examinateur
Paul Zimmermann Directeur de recherche, Inria	Rapporteur & examinateur
Patricia Bouyer Directrice de recherche, CNRS	Examinatrice
Xavier Leroy Professeur, Collège de France	Examinateur
Micaela Mayero Maîtresse de conférences, Université Paris-Nord	Examinatrice
Guillaume Melquiond Chargé de recherche, Inria	Directeur de thèse
Pascal Cuoq Directeur scientifique, TrustInSoft	Coencadrant

Thèse de doctorat

NNT: 2020UPASG023

Acknowledgments

My most grateful thanks go to Guillaume and Pascal, who supervised this work. They provided me with the ideal mix of guidance, challenge and freedom throughout these four years of academic and personal growth.

I would also like to thank Karthikeyan Bhargavan and Paul Zimmermann, who reviewed this document and whose remarks greatly enriched it. I also thank Patricia Bouyer, Catherine Dubois, Xavier Leroy, and Micaela Mayero for taking part in the jury.

Finally, I would like to thank the long list of people who supported me during the realization of this work: my colleagues at TrustInSoft, past and present members of the VALS team, my friends and family, and of course Diane.

Contents

List of Figures

Synthèse

Cette thèse traite de la vérification déductive de programmes d'arithmétique entière en précision arbitraire. Cette arithmétique est utilisée dans des contextes où les performances et la correction des algorithmes sont critiques, comme la cryptographie ou les logiciels de calcul formel. Notre cas d'étude est GMP, une bibliothèque d'arithmétique en précision arbitraire très utilisée. Elle propose des algorithmes de pointe écrits en C et en assembleur qui sont très complexes, et optimisés parfois au détriment de la clarté du code. De plus, certaines branches du code sont visitées avec une probabilité très faible (telle que $1/2^{64}$ en supposant des entrées uniformément réparties), ce qui rend difficile la validation du code par des tests. La vérification formelle de GMP est donc un objectif à la fois désirable et difficile.

Notre outil pour cette vérification est la plateforme de vérification déductive Why3. Why3 fournit un langage de programmation et spécification appelé WhyML. C'est un langage fonctionnel apparenté à OCaml. À partir d'un programme WhyML et de sa spécification, Why3 génère un ensemble de formules logiques appelées obligations de preuve, dont la validité implique que le programme satisfait sa spécification. Afin de valider les obligations de preuve, Why3 s'interface avec un grand nombre de prouveurs externes qui peuvent être des solveurs SMT (comme Alt-Ergo ou Z3) ou encore des assistants de preuve tels que Coq ou Isabelle.

Les programmes que Why3 permet de vérifier sont écrits en WhyML mais la bibliothèque GMP est écrite en C. La première contribution de cette thèse est un ensemble d'additions que j'ai faites à Why3 pour permettre la vérification de programmes C. J'ai développé en WhyML un modèle du langage C qui inclut une axiomatisation de sa gestion de la mémoire ainsi que des modèles de divers types de données. J'ai aussi ajouté à Why3 un mécanisme d'extraction qui permet de compiler les programmes WhyML vers du C idiomatique. (Auparavant, le mécanisme d'extraction de Why3 n'offrait qu'OCaml comme langage cible.) À l'aide de ces deux nouveaux outils, les utilisateurs de Why3 peuvent maintenant vérifier des programmes C par extraction. La méthodologie est la suivante. Tout d'abord, on transcrit le programme C à vérifier dans le langage WhyML. À l'aide de mon modèle du C, il est possible d'écrire des programmes WhyML dans un style si proche du C que cette traduction est souvent transparente. Ensuite, à l'aide de Why3 et des méthodes de preuve usuelles, on vérifie le programme WhyML obtenu. Enfin, on compile ce programme vers C en utilisant le nouveau mécanisme d'extraction. On obtient ainsi du code C vérifié.

Parallèlement au développement de ces outils, je les ai utilisés pour vérifier un fragment significatif de la bibliothèque GMP. Le résultat est une bibliothèque

C formellement vérifiée nommée WhyMP qui implémente des algorithmes de pointe issus de GMP, en préservant presque toutes les optimisations et astuces d'implémentation du code d'origine. WhyMP est compatible avec GMP, et ses performances sont comparables. Elle reprend une grande partie de la couche de GMP qui gère les entiers naturels, appelée `mpn`, ainsi qu'une plus petite partie de la couche appelée `mpz` qui contient des fonctions wrappers pour gérer les entiers signés. WhyMP est disponible à l'adresse `https://gitlab.inria.fr/why3/whymp/`.

La vérification de WhyMP a été plus difficile que prévu. En effet, beaucoup d'obligations de preuve impliquant de l'arithmétique non-linéaire se sont avérées peu adaptées aux prouveurs automatiques à ma disposition. J'ai donc dû ajouter de nombreuses annotations au code WhyML pour mener à bien les preuves, y compris pour des buts qui ne semblaient pas particulièrement difficiles de prime abord. Afin d'augmenter le degré d'optimisation de mes preuves, j'ai ajouté à Why3 un mécanisme de preuves par réflexion. Il est maintenant possible d'utiliser des programmes WhyML classiques en tant que procédures de décision. À l'aide d'une telle procédure, j'ai pu supprimer des centaines de lignes d'annotation du code de WhyMP et les remplacer par des preuves automatiques.

La section 1 de cette synthèse présente une version simplifiée de mon modèle mémoire du C. La section 2 explique brièvement le mécanisme de preuves par réflexion, et la section 3 l'illustre sur une preuve d'une fonction de GMP. Enfin, la section 4 donne un aperçu de la bibliothèque WhyMP dans son ensemble.

1 Vérification de programmes C avec Why3

Cette section présente la modélisation en WhyML de la gestion de la mémoire du C. Il inclut une axiomatisation des pointeurs du C, ainsi que quelques fonctions de gestion de la mémoire telles que `malloc` et `free`. Commençons par présenter les contraintes qui ont guidé mes choix techniques.

Une contrainte importante est que le modèle mémoire (ainsi que le mécanisme d'extraction) fait partie de la base de confiance, n'étant pas lui-même vérifié formellement. Par conséquent, il doit être aussi simple que possible afin de minimiser les erreurs. Idéalement, le modèle doit être si simple qu'il est possible d'être convaincu qu'il est correct par simple relecture du code, ou au moins suffisamment simple pour qu'une preuve papier soit possible.

Par ailleurs, il est souhaitable que le code C extrait soit performant. La traduction de WhyML vers C ne doit pas introduire de clôtures superflues ou d'indirections qui ne sont pas explicitement présentes dans le code WhyML. Idéalement, le développeur du code WhyML doit pouvoir prédire le comportement du futur code C extrait par simple analyse du source WhyML, sans avoir besoin de connaître le mécanisme d'extraction en détail. Par conséquent, il est souhaitable que la traduction de WhyML vers C soit aussi transparente que possible.

Nous pouvons maintenant présenter le modèle mémoire proprement dit. Un extrait simplifié est représenté en figure 1. Lors de l'extraction de WhyML vers C, les fonctions du modèle sont remplacées directement par leurs équivalents C, indiqués en commentaire sur la figure.

La tas du C est représenté comme un ensemble de blocs mémoire appelés

```
type ptr 'a = abstract { data : array 'a ; offset : int }

val incr (p:ptr 'a) (ofs:int32) : ptr 'a                              (* p+i
      *)
  requires { 0 ≤ p.offset + ofs ≤ p.data.length }
  ensures { result.offset = p.offset + ofs }
  ensures { result.data = p.data }
  alias { p.data with result.data }

val get (p:ptr 'a) : 'a                                                (* *p
      *)
  requires { 0 ≤ p.offset < p.data.length }
  ensures { result = p[p.offset] }

val set (p:ptr 'a) (v:'a) : unit                                   (* *p = v
      *)
  requires { p.min ≤ p.offset < p.max }
  ensures { p.data = (old p.data)[p.offset ← v] }
  writes { p.data.elts }

val malloc (sz:uint32) : ptr 'a                    (* malloc (sz * sizeof('a))
      *)
  ensures { result.data.length = 0 ∨ result.data.length = sz }
  ensures { result.offset = 0 }

val free (p:ptr 'a) : unit                                         (* free(p)
      *)
  requires { p.offset = 0 }
  ensures  { p.data.length = 0 }
  writes   { p.data }
```

Figure 1: Un modèle mémoire simplifié du C en Why3.

objets dans le standard C. Le type WhyML polymorphe `ptr 'a` représente les pointeurs vers des blocs contenant des données de type `'a`. Le champ `data` d'un pointeur est un tableau qui stocke le contenu du bloc, tandis que le champ `offset` indique quelle celule du bloc est pointée. Cette construction supporte l'aliasing entre pointeurs. En effet, plusieurs pointeurs peuvent avoir le même champ `data`, ce qui signifie qu'ils pointent vers le même bloc mémoire. Grâce au système de types à régions de WhyML, toute affectation à travers un pointeur est répercutée dans tous ses alias. Le mot-clé `abstract` signifie que les champs du type `ptr 'a` sont fantômes et privés; le code client ne peut interagir avec qu'à travers les primitives du modèle mémoire.

La fonction `incr` renvoie la somme d'un pointeur et d'un entier. Comme le prévoit le standard C, il est permis de calculer un pointeur qui pointe soit à l'intérieur d'un bloc valide, soit sur l'élément juste après la fin du bloc. Le mot-clé `alias` déclare que la valeur de retour de `incr` est en alias avec le pointeur pris en argument. Plus précisément, il unifie les régions de `p.data` et `result.data`.

Ceci permet d'écrire une spécification particulièrement courte pour `free`. En effet, l'écriture de `free` dans le champ `data` de son argument induit un effet dit de *reset*. Ceci signifie que la région qui était pointée par `p` devient inaccessible par les alias de `p`, ces derniers devenant invalidés. Ainsi, après un appel à `free p`, les alias de `p` deviennent inutilisables.

Ce modèle mémoire simplifié constitue un point de départ qui a été amené à évoluer pour pouvoir exprimer une plus grande variété de programmes qui sont présents dans GMP, mais que le système de types de Why3 ne permet pas d'écrire *a priori*. Le modèle complet est bien plus complexe, mais il permet d'écrire des fonctions qui prennent en argument deux pointeurs qui pointent vers des sections séparées d'un objet mémoire. Il est également possible de l'utiliser pour modéliser des fonctions dont les arguments peuvent ou non être en alias.

2 Preuves par réflexion

Afin d'automatiser des preuves simples mais fastidieuses issues des algorithmes de GMP, j'ai ajouté à Why3 un mécanisme de preuves par réflexion. Le principe général des preuves par réflexion est le suivant. Supposons qu'on cherche à prouver une proposition logique P. La première étape est d'intégrer P au fragment logique de WhyML. Notons $\ulcorner P \urcorner$ le terme résultant. Par exemple, $\ulcorner P \urcorner$ pourrait être l'arbre de syntaxe abstraite de P. Ensuite, on prouve que si $\ulcorner P \urcorner$ satisfait une certaine propriété φ, alors P est valide. Ainsi, pour prouver P, il suffit de vérifier que $\varphi(\ulcorner P \urcorner)$ est valide. Si φ est conçue pour pouvoir être validée par simple calcul, nous avons une procédure de preuve par réflexion.

Malheureusement, la fonction $\ulcorner\urcorner$ n'est pas représentable dans la logique, et $\ulcorner P \urcorner$ peut être un terme tellement grand qu'on ne doit pas s'attendre à ce que l'utilisateur le fournisse manuellement. Il est donc nécessaire de proposer une façon de calculer $\ulcorner P \urcorner$ à partir de P. Ce processus est appelé *réification*. Une approche classique est d'exprimer $\ulcorner\urcorner$ dans un méta-langage du système formel considéré (comme Ltac dans le cas de Coq). Cependant, cette approche demande à l'utilisateur d'apprendre le fonctionnement interne du système, contrairement à l'approche que nous proposons. Nous nous reposons sur les faits suivants. Premièrement, la réciproque de $\ulcorner\urcorner$ est exprimable dans la logique de Why3. Deuxièmement, afin de prouver quoi que ce soit d'utile à l'aide de φ, l'utilisateur doit définir quelque chose s'approchant de la réciproque de $\ulcorner\urcorner$ dans la spécification de φ. Dans l'exemple de la figure 2, il s'agit de la fonction `interp`. Il suffit d'inverser à la volée cette réciproque pour générer $\ulcorner P \urcorner$. Cette approche est similaire à la tactique Coq `quote`, avec quelques améliorations: la fonction d'interprétation peut être plus complexe, les quantificateurs sont supportés, ainsi que la réification du contexte logique. Enfin, l'utilisateur n'a pas besoin de spécifier quels termes sont des constantes.

Le synopsis d'une preuve par réflection en Why3 est donc le suivant. Étant donné un but logique P et une procédure de décision φ, l'utilisateur peut tenter de prouver P par réflexion en utilisant la commande `reflection_f` φ dans l'interface utilisateur. Why3 utilise la spécification de φ pour inverser `interp` et deviner un terme $\ulcorner P \urcorner$ approprié, puis évalue $\varphi(\ulcorner P \urcorner)$. La spécification de φ est ensuite utilisée comme indication de coupure pour tenter de prouver P. Dans l'exemple de la figure, si $\varphi(\ulcorner P \urcorner) =$ `True`, on prouve P facilement si $\ulcorner P \urcorner$ a été bien choisi. Si $\ulcorner P \urcorner$ a été mal choisi, ou encore si $\varphi(\ulcorner P \urcorner) =$ `False`, la preuve échoue. En aucun cas cette approche ne permet de prouver quelque chose de faux, à condition que la correction de φ ait été vérifiée auparavant. Ainsi, si la réification devine une mauvaise valeur pour $\ulcorner P \urcorner$, le pire qui puisse arriver est de ne rien prouver de nouveau.

Habituellement, les procédures de preuve par réflexion sont écrites dans le

```
type t = Var int | And t t | ...
type vars = int → bool

function interp (x:t) (y:vars) : bool =
match x with
  | Var n → y n
  | And x1 x2 → interp x1 y && interp x2 y
  ...
end

let φ (x:t) : bool
  ensures { result → forall y. interp x y }
```

Figure 2: Example specification for φ

langage logique d'un système de preuve. Cependant, cela restreint leur expressivité. Ainsi, les fonctions logiques de Why3 ne peuvent pas avoir d'effets de bord et leur terminaison doit être prouvée. Au contraire, avec notre approche, les procdures de décision sont des programmes WhyML comme les autres. Elles peuvent utiliser toutes les fonctionnalités impératives du langage, comme les tableaux, références et les exceptions. Leur correction est prouvée avec Why3, et leur contrat est utilisé comme indication de preuve. Cependant, elles n'ont pas d'implémentation dans la logique de Why3 et ne peuvent pas être interprétées par les prouveurs automatiques. J'ai donc ajouté à Why3 un interpréteur de programmes WhyML. Il opère sur un langage intermédiaire du mécanisme d'extraction qui correspond aux programmes WhyML dont les assertions et le code fantôme ont été effacés. Cet interpréteur nous permet d'exécuter les procédures de décision. C'est la seule partie du mécanisme de preuve par réflexion qui fait partie de la base de confiance. Cependant, comme il se repose sur un langage intermédiaire existant, son implémentation est très simple et l'extension de la base de confiance est minime.

3 Exemple de preuve: multiplication par un limb

La section suivante présente la preuve complète d'une fonction de GMP. L'assertion la plus difficile est prouvée par réflexion.

Commençons par expliquer la représentation des grands nombres dans la bibliothèque GMP. Les entiers naturels y sont représentés par des tableaux d'entiers machine (typiquement de 64 bits) appelés *limbs*. Posons une base $\beta = 2^{64}$. Tout entier naturel $N < \beta^n$ admet une décomposition $\sum_{k=0}^{n-1} a[k]\beta^k$ en base β. On représente N par le tableau $a[0]a[1]\dots a[n-1]$, en commençant par les limbs les moins significatifs.

Cette représentation des grands entiers n'inclut pas de champ pour stocker la taille du tableau. Les opérandes des fonctions bas niveau de GMP sont spécifiées par une paire d'arguments: un pointeur vers le limb le moins significatif, et un nombre de limbs (de type `int32` dans ce document). Introduisons une notation pour la valeur d'un entier représenté par un pointeur et un nombre de limbs. Si un pointeur a est valide sur une longueur n, on note $\texttt{value}(a, n) = \sum_{k=0}^{n-1} a[k]\beta^k$.

On peut maintenant s'intéresser à la preuve d'un algorithme de GMP qui multiplie un entier par un limb. La figure 3 montre l'implémentation de GMP, avec des changements minimes destinés à rendre le code plus lisible. Ma transcription en WhyML est représentée en figure 4.

```
mp_limb_t mpn_mul_1 (mp_ptr rp,
    mp_srcptr up, mp_size_t n,
    mp_limb_t vl)
{
  mp_limb_t ul, cl, hpl, lpl;
  cl = 0;
  do {
      ul = *up++;
      umul_ppmm (hpl, lpl, ul, vl);
      lpl += cl;
      cl = (lpl < cl) + hpl;
      *rp++ = lpl;
  } while (--n != 0);
  return cl;
}
```

Figure 3: Multiplication d'un entier par un limb : implémentation de GMP

La transcription de l'algorithme en WhyML est un procédé relativement simple. Les variables C sont traduites en références WhyML. Cependant, certaines constructions du C ne peuvent pas être traduites directement. C'est le cas de la boucle do-while et des opérateurs de pré- et post-incrémentation. De plus, les arguments de la fonction doivent être copiés dans des références. Cependant, le code extrait (montré en figure 5) a des performances similaires à celles de l'implémentation d'origine.

L'algorithme proprement dit est très simple. L'invariant principal est le suivant :

$$\texttt{value}(r, i) + \beta^i \times c_l = \texttt{value}(x, i) \times y.$$

On multiplie chaque limb de x par y, et on ajoute les résultats à r après décalage. La primitive de multiplication utilisée, `mul64_double`, fait partie du modèle axiomatique des entiers machine. Elle multiplie deux entiers de 64 bits et renvoie le produit (de 128 bits) sous forme d'une paire de mots. Le mot le plus significatif de chaque résultat de multiplication est stocké en tant que retenue dans la variable cl. Pour simplifier les invariants de boucle, on introduit la variable fantôme i. Elle contient le décalage entre la valeur courante des pointeurs r et x (incrémentés à chaque tour de boucle) et leurs valeurs de départ.

Les solveurs automatiques n'ont aucun mal à prouver que les invariants de boucle sont initialisés correctement et qu'ils suffisent à prouver les postconditions de la fonction. La partie la plus difficile de la vérification de cette fonction est de prouver que les invariants sont maintenus entre chaque itération de la boucle. Ainsi, de nombreuses annotations sont nécessaires dans le corps de la boucle. Par exemple, l'assertion des lignes 28–30 sert à prouver que l'addition de `h` et 1 qui peut se produire en ligne 33 ne peut pas déborder.

Notons que le code utilise deux primitives d'addition distinctes qui sont toutes deux remplacées par l'opérateur `+` du C après extraction. La fonction `add_mod` (ligne 31) correspond au cas général de l'addition de deux entiers non signés en C. Elle prend en argument deux entiers machines et renvoie la somme modulo $\texttt{radix} = 2^{64}$. L'opérateur `(+)`, utilisé en ligne 33, est plus restrictif. Il

```
let wmpn_mul_1 (r x:ptr uint64) (y:uint64) (sz:int32) : uint64
  requires { valid x sz }
  requires { valid r sz }
  ensures { value r sz + (power radix sz) * result = value x sz * y }
  ensures { forall j. (j < offset r ∨ offset r + sz ≤ j) →
              r.data.elts[j] = old r.data.elts[j] }
  writes { r.data.elts }
=
  let ref cl = 0 in
  let ref ul = 0 in
  let ref n = sz in
  let ref up = C.incr x 0 in
  let ref rp = C.incr r 0 in
  let ghost ref i : int32 = 0 in
  while n ≠ 0 do
    invariant { 0 ≤ n ≤ sz }
    invariant { i = sz - n }
    invariant { value r i + (power radix i) * cl = value x i * y }
    invariant { rp.offset = r.offset + i }
    invariant { forall j. (j < offset r ∨ offset r + sz ≤ j)
                  → (pelts r)[j] = old (pelts r)[j] }
    ...
    variant { n }
    label StartLoop in
    ul ← C.get up;
    up ← C.incr up 1;
    let l, h = mul_double ul y in
    assert { h < radix - 1
             by ul * y ≤ (radix - 1) * (radix - 1)
             so radix * h ≤ ul * y };
    let lpl = add_mod l cl in
    begin ensures { lpl + radix * cl = ul * y + (cl at StartLoop) }
      cl ← (if lpl < cl then 1 else 0) + h;
    end;
    value_sub_update_no_change (pelts r) (r.offset + int32'int i)
                              r.offset (r.offset + int32'int i) lpl;
    C.set rp lpl;
    assert { value r i = value r i at StartLoop };
    assert { (pelts r)[offset r + i] = lpl };
    value_tail r i;
    value_tail x i;
    assert { value x (i+1) = value x i + power radix i * ul };
    assert { value x (i+1) * y = value x i * y + power radix i * (ul * y) };
   (* nonlinear *)
    assert { value r (i+1) + power radix (i+1) * cl = value x (i+1) * y }; (*
    by reflection *)
    rp ← C.incr rp 1;
    n ← n-1;
    i ← i+1;
  done;
  cl

```

Figure 4: Multiplication d'un entier par un limb : transcription en WhyML

```
uint64_t wmpn_mul_1(uint64_t * r, uint64_t * x, int32_t sz, uint64_t y) {
  uint64_t cl, ul;
  int32_t n;
  uint64_t * up;
  uint64_t * rp;
  uint64_t l, h, lpl;
  struct __mul64_double_result struct_res;
  cl = UINT64_C(0);
  ul = UINT64_C(0);
  n = sz;
  up = x + 0;
  rp = r + 0;
  while (!(n == 0)) {
    ul = *up;
    up = up + 1;
    struct_res = mul64_double(ul, y);
    l = struct_res.__field_0;
    h = struct_res.__field_1;
    lpl = l + cl;
    cl = (lpl < cl) + h;
    *rp = lpl;
    rp = rp + 1;
    n = n - 1;
  }
  return cl;
}
```

Figure 5: Multiplication d'un entier par un limb : code C extrait

prend en argument deux entiers non signés dont la somme est inférieure à `radix` (pas de débordement) et renvoie leur somme.

Le corps de la boucle contient aussi des appels à des fonctions-lemmes (lignes 35, 40, 41). Il s'agit de fonctions qui n'ont pas de contenu exécutable, mais dont la spécification tient lieu de lemme. Ces appels sont effacés par l'extraction. L'appel au lemme `value_sub_update_no_change` (lignes 35–36) établit le fait que l'écriture dans $(r + i)$ (ligne 37) ne changera pas $\texttt{value}(r, i)$. Les appels au lemme `value_tail` (lignes 40 et 41) décomposent les valeurs de `r` et `x` de la manière suivante :

$$\texttt{value}(r, i + 1) = \texttt{value}(r, i) + \beta^i \times r[i],$$

$$\texttt{value}(x, i + 1) = \texttt{value}(x, i) + \beta^i \times x[i].$$

Après la ligne 42, on souhaite prouver l'invariant de boucle principal, et le contexte logique et le but sont essentiellement comme suit:

```
axiom H: value r1 i + (power radix i * cl1) = value x i * y
axiom H1: lpl + radix * cl = ul * y + cl1
axiom H2: value r i = value r1 i
axiom H3: value r (i+1) = value r i + power radix i * lpl
axiom H4: value x (i+1) = value x i + power radix i * ul
goal g: value r (i+1) + power radix (i+1) * cl = value x (i+1) * y
```

La procédure de décision que j'ai développée pour éliminer des assertions de mes preuves de GMP utilise le pivot de Gauss pour vérifier si le but est une combinaison linéaire d'égalités présentes dans le contexte logique. Les coefficients sont des produits de nombres rationnels et de puissances entières de β dont les exposants peuvent être symboliques. Ici, le but est presque une combinaison linéaire des axiomes, mais pas tout à fait. En effet, on ne peut clairement pas former le terme `value x (i+1) * y` à partir de combinaisons linéaires de termes du contexte. On remédie à cela en ajoutant l'assertion de la ligne 43 qui ajoute au contexte le fait suivant:

```
axiom H5: value x (i+1) * y = value x i * y + power radix i * (ul * y)
```

Cette assertion est facilement prouvée par les solveurs SMT. Elle est identique à l'égalité H4 dont les deux termes ont été multipliés par `y`. Une fois la partie non-linéaire de la preuve ainsi traitée manuellement, la procédure de décision sur les systèmes d'équations linéaires termine la preuve en trouvant essentiellement

$$\mathtt{H3} + \mathtt{H2} + \beta^{\mathtt{i}}\mathtt{H1} + \mathtt{H} - \mathtt{H5} \Rightarrow \mathtt{g}$$

ce qui prouve l'assertion de la ligne 44 et l'invariant de boucle principal.

4 WhyMP

Après extraction vers C, l'ensemble de mes algorithmes forme une bibliothèque efficace et formellement vérifiée appelée WhyMP. Elle va bien au-delà des bibliothèques d'arithmétique en précision arbitraires vérifiées existantes en termes de quantité et de qualité des algorithmes. À ma connaissance, il s'agit aussi du développement Why3 le plus ambitieux à ce jour en terme de taille et d'effort de preuve.

WhyMP fournit les algorithmes suivants sur les entiers naturels, issus de la couche `mpn` de GMP : les quatre opérations arithmétiques élémentaires, des décalages logiques, une multiplication et une racine carrée de type diviser-pour-régner, une exponentiation modulaire rapide, et des fonctions de conversion entre la représentation des entiers de GMP et des chaînes de caractères encodant les nombres dans une base arbitraire. Elle contient aussi des fonctions wrapper issues de la couche `mpz` de GMP qui permettent de calculer les opérations élémentaires sur les nombres relatifs en appelant les fonctions sur les entiers naturels et en traitant les signes à part.

La multiplication présentée dans la section précédente est un exemple simple, mais la plupart des autres fonctions sont bien plus compliquées mathématiquement et algorithmiquement. Par exemple, la division longue utilise une primitive qui divise un entier de trois limbs par un entier de deux limbs à l'aide d'un pseudo-inverse précalculé. L'exponentiation modulaire utilise la réduction de Montgomery pour calculer modulo une puissance de 2 plutôt qu'un nombre premier. Enfin, le cas de base de la racine carrée utilise de l'arithmétique à virgule fixe pour calculer la racine carrée d'un entier machine par la méthode de Newton.

Les sources de WhyMP totalisent environ 22 000 lignes de code WhyML, pour environ 5 000 lignes de code C extrait. Le détail de l'effort de preuve est représenté en figure 6.

Parmi les 22 000 lignes de code dans les sources de WhyMP, environ 8 000 lignes sont des instructions exécutables. Les 14 000 lignes restantes sont composées de spécifications et surtout d'assertions. Ce rapport entre la quantité de preuves et la quantité de code est plutôt inefficace par rapport aux développements présents dans le répertoire d'exemples de Why3. En effet, de nombreuses

comparaison	100
addition	1000
soustraction	1000
mul (naïve)	700
mul (Toom-Cook)	2400
division	4500
lemmes auxiliaires	300
procédure de décision	1700
décalages logiques	1000
racine carrée	1600
exponentielle modulaire	1700
conversions	2000
couche `mpz`	3600
utilitaires	200

Figure 6: Effort de preuve en lignes de code.

preuves de propriétés arithmétiques complexes sont effectuées à l'aide de très longues assertions (parfois jusqu'à une centaine de lignes), bien que certaines aient pu être remplacées par des preuves automatiques par réflexion.

L'existence de WhyMP augmente la confiance qu'on peut avoir dans GMP, mais n'est pas une preuve formelle de la correction du code d'origine. Cependant, le développement de WhyMP a permis la découverte d'un bug dans la fonction de comparaison d'entiers relatifs de GMP. Une preuve de non-débordement de retenue dans l'algorithme de multiplication Toom-Cook s'est aussi avérée suffisamment difficile pour convaincre les développeurs de GMP de modifier le code afin de le rendre plus clairement correct.

Le code de WhyMP est suffisamment semblable à celui de GMP pour que les deux bibliothèques soient compatibles entre elles et que leurs performances soient similaires. Il y a deux différences principales. Premièrement, pour chaque opération arithmétique, GMP implémente de nombreux algorithmes qui la calculent. Par exemple, il y a plus d'une dizaine de fonctions de multiplication différentes dans GMP. En fonction de la taille des opérandes, GMP choisit l'algorithme le plus efficace. Ainsi, pour les nombres de moins de 1 000 bits environ (le seuil exact dépend de l'architecture), l'algorithme naïf est utilisé. Pour les nombres d'entre 1 000 et 100 000 bits, GMP utilise un algorithme de Toom-Cook qui est également présent dans WhyMP. En revanche, pour les nombres encore plus grands, GMP utilise des algorithmes avec une meilleure complexité asymptotique qui ne sont pas implémentés dans WhyMP. GMP est donc bien plus efficace que WhyMP sur les entrées de très grande taille.

La seconde différence vient des primitives arithmétiques. En effet, les algorithmes de GMP reposent sur un petit nombre de primitives arithmétiques qui effectuent des opérations élémentaires sur des mots machine (telles que la multiplication de deux entiers de 64 bits, ou la division d'un entier de 128 bits par un entier de 64 bits). Dans la configuration par défaut de GMP, ces primitives sont écrites directement en assembleur. GMP propose également une configuration portable où les primitives sont écrites en C, au prix d'un ralentissement d'environ 100%. En forçant les deux bibliothèques à utiliser des primitives écrites en C, les performances de WhyMP sont très proches de celles de GMP pour les tailles de nombres où les deux bibliothèques utilisent le même algorithme (de l'ordre de 10 à 20% d'écart en faveur de GMP).

Il serait souhaitable de permettre à WhyMP d'utiliser les primitives assembleur afin de rattraper les performances de la configuration par défaut de GMP. Afin de ne pas trop étendre la base de confiance de notre bibliothèque vérifiée, une perspective de travail serait d'étendre Why3 afin de permettre la vérification de code assembleur pour les architectures les plus courantes. On pourrait imaginer procéder de la même manière que pour le C, c'est-à-dire écrire du code WhyML très semblable structurellement à du code assembleur au moyen d'un modèle mémoire, puis compiler vers de l'assembleur.

Une autre amélioration serait d'ajouter à WhyMP les fonctions qui lui manquent par rapport à GMP, comme les fonctions de théorie des nombres (calcul efficace du PGCD, symbole de Legendre...). GMP fournit aussi des versions des opérations élémentaires et de l'exponentielle modulaire destinées à la cryptographie. Ces fonctions sont conçues pour être résistantes aux attaques par canaux cachés. Il serait intéressant de vérifier leur correction, et idéalement, de trouver un moyen de spécifier formellement et de prouver cette résistance.

Chapter 1

Introduction

As the saying goes, every non-trivial program contains at least one bug. Software defects are as ubiquitous as software itself. In his 2004 handbook *Code Complete* [69], Steve McConnell reports “about 15-50 errors per 1000 lines of delivered code” as an industry average. While high-profile security vulnerabilities such as Spectre and Meltdown or the Heartbleed bug make it to the mainstream news cycle every few years, an astonishing number of bugs exist in deployed code. The Common Vulnerabilities and Exposures (CVE) database lists over twelve thousand publicly known security vulnerabilities for the year 2019 alone. A report from the Consortium for IT Software Quality estimates the cost of poor quality software at over two trillion dollars in the United States for the year 2018 [59].

Software bugs also kill. A classical example is the bug in the Therac-25 radiation therapy system, which administered lethal overdoses of radiation to at least five patients between 1985 and 1987 due to concurrent programming errors [64].

Formal verification. From human casualties to financial losses, the cost of software bugs is extremely large. As a result, efforts to increase the reliability of software are warranted. The first resort tends to be *testing* [76]. Testing is the process that consists in executing the program in various contexts with the intent to find defects. Software testing is a field of research by itself and a very large number of approaches to testing exist. However, most of them are not exhaustive. Their goal is to find bugs, not to prove that they do not exist.

As a complementary approach to testing, *static analysis* is part of the effort to write correct programs. It consists in analyzing programs without executing them. It encompasses a wide set of specific approaches. A commonly used form of static analysis is *linting*, which consists in analyzing a program’s source code syntactically looking for programming errors, typically in the form of stylistic errors or suspicious constructs.

A fundamental distinction among static analysis frameworks is *soundness*. Sound tools aim to give formal guarantees about the behavior of the programs they analyze. This takes the form of a specification such as “if the analyzer finds no issues, then the analyzed program is correct”, for some notion of correctness. Many widely used static analyzers, such as Facebook’s Infer tool, are not sound, and simply try to find as many bugs as possible. But in some critical contexts, this is not enough, and sound tools are required to ensure the absence of software defects. *Formal verification* is the process of checking the correctness of programs with respect to a formal specification.

As written above, every sound static analysis tool has some soundness theorem stating that programs in which no issues are found are correct. The word "correct" can have very different meanings depending on the tool. Many tools aim to prove the absence of runtime errors such as program crashes, arithmetic overflows or memory corruption. This is the case of most *abstract interpretation* frameworks. Abstract interpretation consists in overapproximating the behavior of a computer program in an effort to verify that some unwanted behaviors do not occur. For example, the values of each numeric variable of the program might be overapproximated by an interval, with the formal guarantee that the concrete value of the variable lies in that interval. This is enough to prove some properties on the analyzed program. For example, it is enough to know that the interval contains only positive values to prove that it is safe to divide by the variable. Successful applications of abstract interpretation in the industry include the verification of the absence of runtime errors in the flight control systems of various Airbus planes using the Astrée static analyzer [24].

Some tools aim to do more than prove the absence of runtime errors. A more ambitious goal is to prove that a program not only has no runtime errors, but computes the right result. The corresponding notion of correctness is called *functional correctness*. It states that the input-output behavior of a program matches its specification. In 1949, Alan Turing published one of the earliest examples of functional correctness verification in a paper titled *Checking a Large Routine* [93]. It starts as follows:

> "How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows."

The method Turing pioneers is a precursor to the now widely-used approach to the verification of functional correctness properties called *deductive verification*. It consists in generating from the program and its specification a set of logical formulas called *proof obligations* or *verification conditions*, and then proving them using deductive reasoning. The validity of these proof obligations implies that the behavior of the system meets its specification. The specification of a program is usually a set of logical formulas called preconditions and another set of formulas called postconditions. The semantics of the specification is that if the preconditions are all met, then the program successfully executes, terminates without runtime error, and the final state of the program satisfies the postconditions. The generated proof obligations are typically proved valid by either automated theorem provers (such as the SMT solvers Alt-Ergo, CVC4 or Z3), or interactive proof assistants (such as Coq, Isabelle or PVS). However, the automated tools do not always manage to prove that a nontrivial program is correct by themselves. In order to prove the verification conditions, the user typically needs to add a number of annotations to the program in the form of loop invariants, assertions, and lemmas. In the end, for this approach to succeed, it is almost necessary that the user understands why their program is correct to the point that they would be able to sketch a proof on paper.

Other approaches than deductive verification can be used to verify functional correctness properties. One such approach is *model checking.* The program is represented as a finite state transition system, and the specifications are properties in *temporal logic.* They can be *safety properties*, which declare that something should never happen, or *liveness properties*, which declare that something should eventually happen. The set of all reachable states is traversed, and for each property, either a counterexample is found (an execution trace that violates the property), or the property is validated. In order to make the set of reachable states more tractable, a commonly used technique is *bounded model checking*, in which one limits the search to executions traces with length shorter than some fixed integer. This allows the problem to be reduced to a propositional satisfiability problem that can be discharged by SAT solvers. CBMC [60] is a well-known model checker for concurrent C programs that has been applied to embedded systems by the industry.

Another approach is to produce code that is correct by construction, instead of verifying it after the fact. *Refinement* consists in starting from a high-level abstract specification and successively deriving lower-level ones until a concrete implementation is reached. The B Method [3] is a method of tool-assisted software development based on this approach. It has been used to develop safety-critical parts of the control systems of the automated lines 1 and 14 of the Paris Métro. The development of the verified microkernel seL4 [56] was performed using a related method. The authors wrote a high-level abstract specification of the kernel's behavior in the Isabelle/HOL theorem prover, as well as an efficient C implementation. They generated an intermediate executable specification from a Haskell prototype. Finally, they used Isabelle to prove that the executable specification was a refinement of the abstract one, and that the C implementation was a refinement of the executable specification.

Computer arithmetic, arbitrary-precision arithmetic. Computer arithmetic is the field that deals with the representation of numbers in computers and in efficient algorithms for manipulating them. Computer arithmetic is ubiquitous in software. I would be hard-pressed to find software that does not perform any arithmetic computation. Moreover, arithmetic bugs are often subtle and hard to detect, yet may have dramatic consequences.

The typical example is the explosion of an unmanned Ariane 5 rocket in 1996, forty seconds after its lift-off, after flipping 90 degrees in the wrong direction. The cause was determined to be a software error in the rocket's inertial reference system [65]. The software attempted to convert a 64-bit floating-point number (corresponding to the rocket's vertical velocity) to a 16-bit signed value. The number was larger than 32 767, the maximal value representable in a 16-bit signed integer. The conversion failed, the software output a debug value which was misinterpreted as actual flight data, and the engines overcorrected accordingly. It is worth mentioning that the buggy software had been successfully used many times on the rocket's predecessor, the Ariane 4. Indeed, that rocket was launched in a different trajectory, with lower terminal velocity, so the overflow never occurred. This illustrates the difficulty inherent in the manual review of arithmetic code, and the way its correctness heavily depends on the usage constraints.

Some bugs do not have immediate consequences, but simply lead systems to fail in exotic situations. For example, a software bug was discovered in 2015 in the electrical system of the Boeing 787 plane. A software counter in the generator control units of the plane would overflow after 248 days of continuous uptime, or 2^{31} hundredths of a second. This would lead to the loss of all electrical power as the control units would go into failsafe mode, resulting in potential loss of control of the aircraft [31].

In both previous examples, the bugs were due to an arithmetic overflow, that is, an attempt to create a value larger than the space allocated to contain it. *Arbitrary-precision arithmetic* software is used to work with numbers larger than the maximum values allowed by the standard hardware number representation. It consists in algorithms and data structures that can be used to represent arbitrarily large numbers, limited only by the memory and time available.

The GMP library. The GNU Multiprecision Arithmetic Library, or GMP for short, is a widely-used library for arbitrary precision arithmetic that operates on integers, rational numbers, and floating-point numbers [45]. It is built with an emphasis on performance. It uses state-of-the-art algorithms written in C, as well as handwritten assembly routines for most common architectures in the most common inner loops. Its main target applications are cryptography algorithms and computer algebra software. It is used as a backend by the computer algebra systems Mathematica and Maple. Compile-time floating-point computations in the GCC compiler suite are also performed by the MPFR library [41], which is itself built on top of GMP.

As with any software development of this size, GMP is not bug-free. While the library is extensively tested, some parts of the code are visited with extremely low probability (such as $1/2^{64}$, assuming uniformly distributed inputs). As a result, some bugs cannot reasonably be found using random testing. Moreover, most of the algorithms are very intricate, which makes both the manual review of the code and the design of edge-case tests challenging. The following is an excerpt from the release news of GMP 5.0.4:

> “Two bugs in multiplication code causing incorrect computation with extremely low probability have been fixed. [...] Two bugs in the gcd code have been fixed. They could lead to incorrect results, but for uniformly distributed random operands, the likelihood for that is infinitesimally small.”

Incorrect results being computed in GMP may even yield security vulnerabilities such as buffer overflows. Consider the following GMP macro.

```
#define mpn_decr_1(x) \
  mp_ptr __x = (x); \
  while ((*(__x++))-- == 0) ;
```

The `mpn_decr_1` macro takes an arbitrary-precision natural integer `x` (represented as an array of machine integers, from least to most significant) and decreases it by one. Note that the `while` loop iterates until it finds a cell in the array with a non-zero value. As a result, if `mpn_decr_1` is called on the integer 0, the macro reads and writes outside the bounds of the array, possibly resulting in memory corruption or crashing the program. The takeaway is that functional correctness and absence of runtime errors are not independent. In order to meaningfully verify that GMP has no runtime errors, one also needs to verify its functional correctness.

However, this is easier said than done. GMP implements state-of-the-art algorithms which can be very intricate. Furthermore, the implementation has been optimized as much as possible, with performance prioritized over clarity. Even the so-called schoolbook algorithms are somewhat obfuscated by layers of optimizations and implementation tricks. Some seemingly innocuous statements require complex proofs in order to show that, for instance, an operation or a carry propagation does not overflow. Textbooks such as Brent and Zimmermann's [17] explain the broad strokes of most of the algorithms, but do not mention most of these implementation tricks in their pseudocode. As a result, understanding why the algorithms are in fact correct is the first challenge to overcome when verifying GMP, and it is not a trivial one.

Motivation. The motivation for this work is twofold. First, the formal verification of a meaningful fragment of GMP is a worthy goal by itself. Indeed, GMP is used in security-sensitive contexts where an incorrect result may be costly. The second motivation is to develop new verification methods using GMP as a case study. GMP's source code presents a particular combination of challenges. It features very intricate mathematical algorithms, but also low-level memory manipulation. While it does not quite exercise C's fine-grained pointer manipulation to its full extent (for example, it uses almost no type punning), it does perform pointer arithmetic and features complex pointer aliasing configurations.

There has been some previous work on the formal verification of arbitrary-precision arithmetic libraries. Myreen and Curello verified an implementation of the four basic arithmetic operations in arbitrary precision using the HOL4 theorem prover and separation logic [77]. Affeldt used the Coq theorem prover to verify an arbitrary-precision GCD algorithm implemented in a variant of MIPS assembly, as well as all the functions it depends on [4]. Fischer verified a modular exponentiation library using Isabelle/HOL [39]. Berghofer used Isabelle/HOL to develop a verified bignum library providing modular exponentiation written in the SPARK fragment of the Ada programming language. However, none of these verified libraries are meant to be used in practical contexts. The algorithms are not as efficient as GMP's state-of-the-art ones, and the implementations are not as optimized.

Shortly after this work began, Zinzindohoué et al. completed the verification of a formally verified cryptography library implementing the full NaCl API with state-of-the-art performance [96]. Their work includes a bignum library, but the algorithms they verified use numbers whose length is known in advance, so they do not pose quite the same verification challenges as arbitrary-precision algorithms.

Intricate arbitrary-precision algorithms have also been the subject of formal verification, including GMP's own divide-and-conquer square root algorithm, which was verified by Bertot et al. using the Coq proof assistant [11]. However, their approach was quite costly in terms of proof effort, and does not seem to scale up to the verification of an entire library in a practical way.

In the end, no verified arbitrary-precision arithmetic library with both the breadth and depth of GMP currently exists. Our ambition is to develop such a library, and push existing verification techniques to their limits in the process.

Tool choice. In order to explain our methodology and tool choice for this verification work, let us first lay out our requirements. We intend to verify a C library that involves *pointer arithmetic and aliasing* in complex ways. As a result, our verification toolset must either have explicit support for the C language, or enable us to design a sufficiently expressive model of the C memory ourselves. However, pointer manipulation is not our only concern. We are attempting to verify the functional correctness of highly intricate arbitrary-precision arithmetic algorithms. As far as I'm aware, no specialized tool to perform exactly this task currently exists. As a result, our toolset must have sufficiently powerful *general-purpose theorem proving* capabilities. Finally, we want to verify a large amount of GMP algorithms, in order to obtain a usable final product. The available proof effort being three to four man-years, our toolset must provide a meaningful amount of *proof automation.* In particular, the *frame problem* (i.e., the knowledge of which memory areas are modified or not by a given instruction) should be solved in a somewhat automated way.

Let us now review some available tools and check them against our criteria. A natural choice for the verification of C programs is the Frama-C platform [25]. It is most commonly used to verify the absence of runtime errors in C programs using the abstract-interpretation Value plugin, as well as its successor EVA. However, it has also been used to verify functional correctness properties in data structure implementations using the deductive verification WP plugin [14]. Using the ACSL specification language [8], comments can be used to specify the intended behavior of the program, state axioms and theorems, and write ghost code. Frama-C computes verification conditions which are then discharged using automated solvers or the Coq theorem prover. Frama-C has a built-in C memory model that is expressive enough to implement GMP's algorithms. However, it does not appear to automate the tracking of aliases or the associated frame problem. Moreover, expressing complex mathematical properties in ACSL comments seems challenging, and proving them using ACSL annotations and automated theorem provers seems out of reach of current tools. As a result, using Frama-C to verify a large fragment of GMP would probably require proving most non-trivial arithmetic facts in Coq, and it is not clear that it would be an improvement over doing the entire verification work in Coq in the first place.

Performing the verification using Coq, or some other interactive proof assistant such as Isabelle/HOL, would also have its merits. These tools can certainly be used to verify arbitrarily complex arithmetic facts. They have already been used to verify C programs, as well as the certified C compiler CompCert [63]. As a result, models and semantics of the C language in Coq exist in the wild and could be used. The main issue with this approach would be the amount of proof effort required. The existing example of verification of a single GMP algorithm by Bertot et al. involved 13 000 lines of Coq. It was performed by experts including a GMP developer. Our objective is to verify a meaningful amount of GMP algorithms, such as twenty or more. While part of the proof effort can certainly be reused and mutualized among the various algorithms to verify, such a proof would still likely take upwards of 100 000 lines of code. There is little reason to believe that this would be feasible by a single student in a few years. Proof automation in Coq has improved since the verification of GMP's square root algorithm, but the existing tools are unlikely to be usable out of the box on the goals that arise from the proofs of GMP algorithms (non-linear arithmetic with symbolic exponents). Developing tactics able to meaningfully automate the proofs of GMP algorithms would take a nontrivial amount of time and effort.

A good way to deal with complex aliasing configurations and automatically instantiate the frame rule could be to use a tool based on separation logic such as Smallfoot [9], Verifast [54], or Viper [75]. However, these tools do not appear to be well-suited to the verification of many complex arithmetic goals. Indeed, while they allow users to state complex specifications and lemmas, they lack ways to enable the user to prove them when the automated solvers fail to do so automatically. In other words, I am not aware of a separation-logic-based tool that has sufficiently powerful general-purpose theorem proving capabilities, outside of the various embeddings of separation logic in interactive theorem provers.

Giving up on built-in separation logic, we move to general-purpose deductive verification platforms such as Why3 [37], Leon [13], Dafny [61], and F* [91]. They all offer a degree of proof automation by interfacing with one or more automated solvers. Several of these tools also seem able to verify pointer-heavy C programs. Why3 features a region-based type system with automated alias tracking [44], and enables users to develop their own memory models. F*'s dependent type system is expressive enough to develop a model of the C memory that deals with aliases in a precise way [81]. The remaining criterion is the ability to prove highly complex goals. Why3 appears to be the best deductive verification tool in this area. It interfaces with a wide set of automated solvers with various strengths and weaknesses [34], whereas most tools only use one. This enables users to call on the best solver for each situation, including highly specialized ones such as the Gappa numerical constraint solver [26, 27]. Furthermore, when all else fails, the user can always fall back to a Coq proof. While we would rather avoid this (and indeed, Coq was only used a handful of times in this work), it is a good insurance that the tool will not get stumped by a particularly difficult arithmetic goal.

Methodology. Locking in Why3 as our tool of choice, let us consider our options in terms of methodology. Why3 enables users to verify programs written in a functional programming language called WhyML. It has been used in the past to verify C programs using various methods. One such method is to use WhyML as an intermediate language. Using a front-end such as the Jessie Frama-C plugin, the C program to verify is translated to WhyML and verified using Why3 [67]. This strategy is quite similar to the Frama-C/WP one described earlier. Ultimately, it is ill-suited to our problem for much the same reasons. Although it uses Why3 as a backend, it cannot leverage its theorem-proving capabilities to their full extent.

Another strategy consists in first reimplementing the C program in WhyML, and then verifying the WhyML implementation using Why3 [33]. WhyML is a functional language, but it is expressive enough that it is possible to axiomatize a model of the C language and implement GMP's programs on top of it. This approach enables us to verify C-like programs while using Why3 to its fullest. However, the obvious drawback is that we would not verify GMP's algorithms, but their abstraction in another language. How to turn this abstraction into a usable verified library? Similarly to Coq, Why3's usual answer to this question is *program extraction.* WhyML programs can be compiled to OCaml code. Extending Why3's extraction mechanism to include C as a target language seemed challenging but doable. This addition to Why3 enables the development of a verified C arbitrary-precision library that has performances comparable to GMP.

Contributions and plan. This document starts with an introductory chapter that explains the basics of Why3 (Chap. 2). It gives a brief overview of WhyML both as a programming language and a specification language, and provides some explanations on Why3's verification condition generator and how it interacts with aliasing. It contains no new contributions and can be skipped by readers familiar with Why3.

One of the objectives of this work was to enable Why3 users to produce verified C programs. I have developed a set of axiomatized models of the C memory and various C datatypes in WhyML. I have also extended Why3's extraction mechanism so that WhyML programs can be compiled to readable and efficient C code, where only OCaml and some of its dialects were previously supported. Thus, Why3 users can implement their programs in WhyML on top of the C model, verify them, and obtain verified C code. Chapter 3 presents these contributions to Why3 and walks through the verification of a small data structure from the open source operating system Contiki.

Concurrently to developing these tools for the verification of C programs, I have used them to verify a sizable fragment of the GMP library. One objective of this formalization work was to evaluate whether the tools I added to Why3 scale up to the verification of real-world programs, as well as inform the development of these tools by uncovering issues that toy examples would not have. However, this verification work is also its own reward.

The end result of this formalization is a verified C library called WhyMP. It implements GMP's state-of-the-art algorithms, with almost all of the original implementation tricks preserved. As a result, it is compatible with GMP and has comparable performance. The extracted C code is about 5 000 lines long and the formalization took about 22 000 lines of WhyML code. WhyMP implements a large subset of the algorithms in GMP's `mpn` layer, which deals with natural integers, as well as a smaller subset of the `mpz` layer, which is a wrapper around `mpn` that deals with relative integers. Chapter 4 presents the formalization of WhyMP's algorithms: the four basic operations, fast modular exponentiation, divide-and-conquer multiplication and square root, base conversions, and the `mpz` wrappers. It also compares WhyMP and GMP in terms of compatibility, exhaustivity, and performance.

The verification of WhyMP was more time-consuming than expected. Many of the non-linear arithmetic facts that needed to be proved were handled poorly by the automated solvers. As a result, many proof annotations were needed, resulting in tedious proof effort even for goals that did not seem conceptually hard. In an effort to increase proof automation, I have added to Why3 a framework for proofs by reflection. Users may now write decision procedures as regular verified WhyML programs and use them to compute proofs of their goals. Chapter 5 discusses this contribution and its use in WhyMP's development.

Finally, Chap. 6 sums up this work's contributions, reflects on some challenges encountered in the process, and lays out possible lines of future work.

Chapter 2

Why3 basics

This short chapter aims at providing the reader with enough background about Why3 to make the rest of the thesis readable. It contains no original contributions and may safely be skipped by readers who are already familiar with Why3.

Why3 is a platform for deductive program verification. It provides a programming and specification language called WhyML. It is a functional language with many syntax elements borrowed from the OCaml language. WhyML programs are annotated with contracts based on Hoare logic [51, 40]. From these contracts, Why3 computes verification conditions that, once proved, guarantee that the WhyML programs have no runtime errors and satisfy their specification. Why3 interfaces with external theorem provers to discharge the verification conditions. WhyML programs can be extracted to correct-by-construction OCaml code.

This chapter provides an overview of the WhyML language and its semantics, as well as how the verification conditions are computed. We first go over the basics WhyML as a programming language (Sec. 2.1). Section 2.2 describes WhyML's logic and its syntax as a specification language. In Sec. 2.3, we go over the process by which Why3 computes verification conditions. Finally, Sec. 2.4 gives a brief overview of how Why3 is used in practice.

2.1 The WhyML programming language

Let us first provide a brief overview of the WhyML language. More details can be found in the Why3 reference manual [1].

2.1.1 Basic syntax

A WhyML source file is a list of modules, and a module is a list of declarations. There are three main kinds of declarations: function declarations, type declarations, and logical declarations. Let us first focus on program functions. The general syntax is `let name [parameters] : return_type = expression`. The return type, as well as the types of the parameters, can sometimes be omitted and inferred by Why3. The following is an example of function declaration. It

[1] `http://why3.lri.fr/doc/index.html`

defines the function `f` as the function that takes an integer `x` and returns the integer `x + 42`.

```
let f (x:int) : int = x + 42
```

Much like in OCaml, any function body is an expression, and there is no notion of statement. (One may choose to view statements as expressions whose return type is `unit`.) Local variables are declared using let-bindings, with the usual syntax `let x = e1 in e2`. This expression computes the expression `e1`, binds the result to the variable `x`, and computes `e2`. The semicolon `;` is the sequence operator. The expression `e1; e2` computes the expressions `e1` and then `e2` in sequence. It can be seen as syntactic sugar for `let _ = e1 in e2`. A partial grammar of expressions follows. New constructs will be introduced throughout this chapter.

⟨*expr*⟩ ::= ⟨*constant*⟩	(integer, real, or string constant)
| `true`	
| `false`	(Boolean constant)
| ⟨*ident*⟩	(identifier)
| ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩	(infix binary operator)
| `let` ⟨*pattern*⟩ = ⟨*expr*⟩ `in` ⟨*expr*⟩	(let-binding)
| ⟨*ident*⟩ ⟨*expr*⟩+	(function call)
| `if` ⟨*expr*⟩ `then` ⟨*expr*⟩ `else` ⟨*expr*⟩	(conditional)
| `(`⟨*expr*⟩ `(,` ⟨*expr*⟩`)+ )`	(tuple)
| `match` ⟨*expr*⟩ `with (|`⟨*pattern*⟩ `->` ⟨*expr*⟩`)+ end`	(pattern matching)
| `while` ⟨*expr*⟩ `do` ⟨*expr*⟩ `done`	(loop)
| `raise` ⟨*ident*⟩	(exception raising)
| `try` ⟨*expr*⟩ `with (|`⟨*pattern*⟩ `->` ⟨*expr*⟩`)+ end`	(exception catching)

Function calls use the same syntax as OCaml. However, while the evaluation order of the arguments is left unspecified in OCaml, it is specified in WhyML. Function arguments are evaluated from right to left. For example, if a function `g` takes two parameters, then the function call `g e1 e2` evaluates first `e2`, then `e1`. The chosen order is arbitrary, but there needs to be one to compute precise verification conditions.

The normalization of WhyML function calls actually goes even further. Although function calls accept arbitrary expressions in the surface language, WhyML programs are transformed before any verification conditions are computed. Function calls are put into A-normal form [87], that is, all function arguments must be trivial. So, if `e1` and `e2` are not simple variables, the function call `g e1 e2` is transformed into `let v2 = e2 in let v1 = e1 in g v1 v2`, as function arguments are evaluated from right to left.

2.1.2 Mutability

Some Why3 objects are mutable. The most simple example is references. Other examples include records with at least one mutable field (Sec. 2.1.3). They are declared with the `let ref` construct. The following example computes the product of two positive integers using addition and references.

```
let mul (x y:int) : int
=
  let ref res = 0 in
  for i = 0 to x-1 do
    res ← res + y
  done;
  res
```

The calling conventions around mutable objects is similar to OCaml. All records and references are implicitly boxed. Essentially, when they are passed to a function, it is actually a pointer to them that is passed by value. The end result is similar to what would happen if all mutable structures were passed by reference. For example, in the following snippet, the modification of `x` in the function `set` is visible in the caller's scope.

```
let set (ref x: int) : unit = x ← 42

let f () : int
=
  let ref x = 0 in
  set x;
  (* x = 42 *)
  x
```

This syntax for references may seem unusual to OCaml programmers, who are used to distinguish a reference `x` and its value `!x`. WhyML's syntax, with no visible distinction between reference and value, is actually syntactic sugar. Internally, references are built-in as a record with a single mutable field called `contents`. They are dereferenced automatically, so that if `x` has type `'a` and is declared with the `ref` keyword, internally `x` has type `ref 'a`, the expression `x <- a` is syntactic sugar for `x.contents <- a`, `f x` is sugar for `f x.contents`, and so on.

2.1.3 Data types

WhyML features an ML-style type system with polymorphic types, algebraic data types, and records that can have mutable fields. The primitive built-in types are integers, reals, booleans, and strings. Tuples are also built-in. The unit type is identified with the 0-ary tuple.

Record types

Record types with named fields may be defined using the following ML-style syntax.

```
type t = { a: int; mutable b: int }
```

Record fields can be mutable, such as the `b` field in the example above. Fields are accessed using the syntax `r.a`, and mutable fields are mutated with the syntax `r.b <- 42`. Records can be constructed with the syntax `{a = 36; b = 55}`.

Arrow types

The arrow type `a -> b` denotes a first-class mapping from values of type `a` to values of type `b`. Values of this type can be declared with the syntax `(fun x -> e)`. Note that these are distinct from program functions that take an argument of type `a` and return a value of type `b`, which is a departure from OCaml. These mappings carry no specifications. Most automated solvers only support them in rather limited ways.

A typical use case for mappings is the `numof` function from Why3's standard library. It counts the number of integers in an interval that satisfy some predicate.

```
(** number of `n` such that `a ≤ n < b` and `p n` *)
let rec function numof (p: int → bool) (a b: int) : int
  variant { b - a }
= if b ≤ a then 0 else
  if p (b - 1) then 1 + numof p a (b - 1)
               else     numof p a (b - 1)
```

Algebraic data types

WhyML features algebraic data types declarations in the same style as OCaml. Such definitions can be recursive. A typical example is the definition of the polymorphic list type from the standard library.

```
type list 'a =
  | Nil
  | Cons 'a (list 'a)
```

Note that the polymorphic type expression is `list 'a`, and not `'a list` like in OCaml.

Range and float types

The built-in `int` type denotes unbounded, mathematic integers. There is no notion of integer overflow. This is useful for specifications and proofs, but when writing executable programs meant to be extracted to languages such as OCaml or C, bounded integers are necessary.

If `a` and `b` are two integer literals, the type declaration `type r = < range a b >` defines an integer type that ranges over the interval `[a, b]`. This declaration automatically introduces the function `r'int`, which projects elements of type `r` back to `int`, as well as the logical constants `r'minInt` and `r'maxInt` which represent constants `a` and `b`. An example is the definition of the type `int32` in the standard library, which denotes 32-bit integers.

```
  type int32 = < range -0x80000000 0x7fffffff >
```

Floating-point numbers are in a similar situation as machine integers. The built-in `real` type is an axiomatization of mathematical reals. Floating-point types can be defined using the declaration `type f = <float eb sb>`, where `eb` and `sb` are two literals. This declares the type `f` of floating point numbers with `eb` exponent bits and `sb` significand bits, as defined in the IEEE-754 floating-point standard [2].

2.2 Logic and specifications

WhyML is not just a programming language. WhyML modules may also contain non-executable logical content such as axioms, lemmas, goals, and so on. In principle, one could completely ignore the programming language part of WhyML and use Why3 purely as a proof assistant. However, Why3's logic is not entirely separate from programs. For example, they interact through program specifications and assertions. Why3's logic is essentially first order, with some support for inductive predicates. In the rest of this document, we only make use of first order logic.

Logical declarations

Let us first give a brief overview of the syntax of logical declarations. WhyML *terms* are the logical equivalent of expressions. The main differences between terms and expressions are that terms accept quantifiers and logical connectives, and are always pure (no side effects). Terms that have the proposition type are called *formulas*. Note that Why3 distinguishes between propositions and Booleans internally, but this distinction is not enforced in the syntax, and the conversions are automatically performed behind the scenes.

WhyML declarations include axioms, lemmas, and goals.

```
⟨decl⟩ ::= ...
  | axiom ⟨ident⟩ : ⟨term⟩
  | lemma ⟨ident⟩ : ⟨term⟩
  | goal ⟨ident⟩ : ⟨term⟩
```

Axioms, lemmas, and goals are syntactically the same. As one may expect, the difference is how they are interpreted in the logic. Terms declared as axioms are added to the logical context for the rest of the module, and do not generate a verification condition (that is, the user does not need to prove them). Lemmas need to be proved first, and then they are added to the logical context. Finally, goals need to be proved but do not add anything to the logical context.

WhyML also allows users to declare logical functions. They use a similar syntax as program functions, but they cannot be used in program expressions. Instead, they can be used in logical terms.

```
⟨decl⟩ ::= ...
  | function ⟨ident⟩ ⟨parameter⟩+ : ⟨type-expr⟩ = ⟨term⟩
  | predicate ⟨ident⟩ ⟨parameter⟩+ = ⟨term⟩
```

Predicates are a special case of functions whose return type is a proposition. For example, the following predicate states that two mappings are equal on an interval.

```
predicate map_eq_sub (a1 a2 : int → 'a) (l u : int) =
  forall i:int. l ≤ i < u → a1[i] = a2[i]
```

Function specifications

Arguably, the main use of Why3 as a verification platform is to specify contracts for program functions and to verify that the functions do indeed fulfill their contracts. Function specifications are a list of clauses. The most common clauses are `requires`, which adds a precondition, and `ensures`, which adds a postcondition. The semantics of these clauses is as follows: if the preconditions are valid when the function is called, then the function has no runtime error and the postconditions are valid at the end of the execution. Let us now give a specification to the `mul` function from Sec. 2.1.2.

```
let mul (x y:int) : int
  requires { x ≥ 0 }
  ensures { result = x * y }
=
  let ref res = 0 in
  for i = 0 to x-1 do
    res ← res + y
  done;
  res
```

The `mul` function computes the product of `x` and `y` if `x` is non-negative, which is specified in the postcondition. If `x` is negative, the function returns 0 (incorrectly unless `y` is 0), so we exclude this case in the precondition. Note that there is no consideration of overflow. Indeed, the built-in `int` type corresponds to unbounded, mathematical integers. If we ask Why3 to verify this function, it fails. Indeed, the proof that `mul` indeed computes `x * y` is not entirely trivial. Essentially, it is a proof by induction on `i`. However, the `for` loop is not unrolled by Why3 (more details can be found in Sec. 2.3). The user needs to give an extra hint to help Why3 prove this function. In this case, this hint is a loop invariant. WhyML `for` and `while` loops can be annotated by one or more loop invariants using a clause such as `invariant {f}`. This clause means that the formula `f` is valid at the start of each iteration of the loop, as well as at the loop exit (in this case, when `i=x`). In our case, a valid invariant is as follows:

```
let mul (x y:int) : int
  requires { x ≥ 0 }
  ensures { result = x * y }
=
  let ref res = 0 in
  for i = 0 to x-1 do
    invariant { res = i * y }
    res ← res + y
  done;
  res
```

This decomposes the task of proving the postcondition into four subtasks:

1) At the start of the loop, `res = 0 * x` (invariant initialization).

2) For all `0 <= n <= x-1`, assuming the invariant is valid for `i = n`, prove that it stays valid for `i = n+1` (invariant preservation).

3) At the end of the loop, knowing that `res = ((x-1) + 1) * y`, prove the postcondition.

4) Assuming that the loop was never entered (that is, `x=0`), prove the postcondition.

All four tasks are easily discharged by automated solvers, and the program is proved correct.

Abstract functions

Program functions may be declared without a body using the `val` keyword. They are the program-side equivalent of axioms. Their specification is accepted without proof.

A typical use case for `val` declarations is the definitions of data structures in WhyML's standard library. The following example comes from the module that define WhyML arrays. It is the equivalent of OCaml's `Array.make`, that is, it creates an array of `n` cells initialized to `v`.

```
val make (n: int) (v: 'a) : array 'a
  requires { n ≥ 0 }
  ensures { forall i:int. 0 ≤ i < n → result[i] = v }
  ensures { length result = n }
```

This function can be called inside WhyML code exactly like a regular program function. Indeed, when computing verification conditions of a program that involves a function call, Why3 does not consider the body of the called function, but only its specification. However, `make` obviously cannot be extracted to OCaml code. Instead, it should be replaced by a suitable OCaml function, such as `Array.make` (Sec. 3.4.4).

Function specifications may also have a `writes` clause that specifies which mutable regions are modified by the function, such as `writes { x, y.data }`. If there is no `writes` clause in the specification of a non-abstract function, it can be inferred automatically by Why3 from the function body.

As the specifications of `val` declarations are accepted as axioms, they are a common source of errors. In particular, note that Why3 has no way of inferring which regions are modified by an abstract function. If the user does not provide a `writes` clause, Why3 assumes that the function does not modify anything. For example, the following abstract function would swap two elements in an array, if its author did not forget the `writes` clause. As written, it instead declares that two elements of the array are equal, which rapidly leads to inconsistencies.

```
val bad_swap (a: array int) (i j:int)
  requires { 0 ≤ i < length a ∧ 0 ≤ j < length a }
  ensures  { a[i] = old a[j] ∧ a[j] = old a[i] }
  (* writes { a } was forgotten *)
```

Termination

By default, Why3 users are required to prove that their programs terminate (or annotate explicitly those that do not). Program termination is undecidable in general, so hints from the user are needed. All recursive functions and `while` loops must be annotated with a `variant` clause. The variant is a program expression that is decreasing at each recursive call or loop iteration on some well-founded order.

The following example is the ancient Egyptian multiplication algorithm. It computes the product of `a` and `b` using only additions, multiplications by 2, and divisions by 2. It is analogous to the square-and-multiply algorithm. The loop variant is the variable `y`. It is non-negative and divided by 2 at each iteration (rounding down), so it strictly decreases until loop exit.

```
let mul (a b: int) : int
  requires { b ≥ 0 ∧ a ≥ 0 }
  ensures  { result = a * b }
=
  let ref x = a in
  let ref y = b in
  let ref z = 0 in
  while y ≠ 0 do
```

```
    invariant { y ≥ 0 }
    invariant { z + x * y = a * b }
    variant   { y }
    if mod y 2 = 1 then z ← z + x;
    x ← 2 * x;
    y ← div y 2
  done;
  z
```

Ghost code

WhyML variables and expressions may be declared *ghost* [35]. Ghost code is present for verification purposes only. It may not affect non-ghost computations, or be used in non-ghost expressions, or write in non-ghost mutable fields. This non-interference is enforced by Why3's type system. As a result, ghost code can safely be erased when executing WhyML code or when extracting it to OCaml or C code.

The following example makes use of ghost code to prove the correctness of Euclide's algorithm, which computes the greatest common divisor of two integers. It is actually an implementation of the extended Euclidean algorithm, that is, it computes the Bézout coefficients of `x0` and `y0` in addition to their greatest common divisor. This is specified in the second postcondition. The last two loop invariants are necessary to prove the second one. However, the instructions and variables that pertain to the computation of the Bézout coefficients `a` and `b` are ghost. If they were erased, we would end up with a correct (although more difficult to prove) implementation of the regular Euclidean algorithm.

```
let gcd (x0 y0: int) : (result: int, ghost a: int, ghost b: int)
  requires { x0 ≥ 0 ∧ y0 ≥ 0 }
  ensures  { result = gcd x0 y0 }
  ensures  { a*x0 + b*y0 = result }
  =
  let ref x = x0 in
  let ref y = y0 in
  let ghost ref a = 1 in let ghost ref b = 0 in
  let ghost ref c = 0 in let ghost ref d = 1 in
  while y > 0 do
    invariant { x ≥ 0 ∧ y ≥ 0 }
    invariant { gcd x y = gcd x0 y0 }
    invariant { a * x0 + b * y0 = x }
    invariant { c * x0 + d * y0 = y }
    variant   { y }
    let r = mod x y in
    let ghost q = div x y in
    x ← y; y ← r;
    let ghost ta = a in
    let ghost tb = b in
    a ← c; b ← d;
    c ← ta - c * q;
    d ← tb - d * q;
  done;
  x, a, b
```

Assertions

Another example of ghost code is program assertions. The expression `assert { f }`, where `f` is a logical formula, requires the user to prove `f` and then adds it to the logical context. It has no computational content. The following is an example of a (trivial) assertion in WhyML code.

```
let x = 42 in
let y = 12 in
assert { x + y = 54 }
```

It may be the case that the formula that is being asserted is itself difficult to prove. In this case, one may give proof cut indications using the `by` and `so` logical connectives [20]. If the formula `A by B` occurs as a goal, Why3 generates two verification conditions: one for `B`, and one for `B -> A`. When it appears as a premise, it is reduced to simply `A`. The keyword `so` is the dual of `by`. When `A so B` appears as a premise, the conditions `A` and `A -> B` are both generated and kept in the context. When it appears as a goal, only `A` is kept. As a result, one may write assertions such as `assert { A by B1 so B2 so ... Bn }`, with as many hints as needed to make the proof easier. The associativity of `by` and `so` is such that the formula is equivalent to `A by (B1 so B2...)`. Why3 generates goals for `B1` and each of the implications in the chain `B1 -> ... -> Bn -> A`. However, after the assertion, only `A` is kept in the logical context, so the proof indications do not pollute the proof context.

Type invariants

Record types can carry invariants, as follows:

```
type t = { a: int; mutable b: int }
         invariant { 0 < a < b }
```

Let us briefly explain the semantics of type invariants. In the logic, type invariants always hold. In particular, in order to construct a value of type `t`, one must first prove that its fields validate the invariant. Only valid records can be passed to logical functions or predicates. In programs, type invariants hold at the boundaries of function calls. They can be temporarily broken in the middle of functions, although they must be restored before passing the record to another function (in order to satisfy the calling convention of the callee). For example, the following function can be verified.

```
let f (x:t) =
  x.b ← 0;       (* invariant broken *)
  x.b ← x.a + 2 (* invariant restored *)
```

However, the following functions cannot. In both cases, the program attempts to pass a broken record to a function or a predicate. Why3 checks the type invariant, which cannot be proved.

```
let g (x:t) =
  x.b ← 0; (* invariant broken *)
  f x;      (* broken record passed to program function *)
  x.b ← x.a + 2

predicate p (x:t) = (x.b ≤ x.a + 1)

let h(x:t) =
  x.b ← x.a;
  assert { not p x }; (* broken record passed to predicate *)
  x.b ← x.b + 1;
```

Snapshots and labels

If `t` is a WhyML type, the snapshot type `{t}` denotes a ghost value of type `t` produced by a pure logical function. As their name indicates, snapshots cannot be mutated. For example, `{int}` is the same as `int`, but `{array int}` denotes an array that can be read from, but not written into. The expression `pure {e}` computes a ghost snapshot of `e`. The following toy program uses it to express its loop invariant more concisely.

```
let ref x = 42 in
let x0 = pure { x } in
for i = 0 to 10 do
  invariant { x = x0 + i }
  x ← x + 1;
done
```

Snapshots may also be accessed retroactively in assertions and specifications using *labels*. Labels are a way to give a name to a particular program point. They are set using the syntax `label L in`, where `L` is the name of the label. The term `e at L` then refers to a snapshot of the expression `e` at the program point labeled by `L`. As an example, the previous snippet is equivalent to the following one:

```
let ref x = 42 in
label StartLoop in
for i = 0 to 10 do
  invariant { x = x at StartLoop + i }
  x ← x + 1;
done
```

Function declarations typology

There are many different kinds of WhyML function declarations. Let us recap them briefly.

`let`

Regular program function with specification and body.

`val`

Program function without a body (specification only).

`function`

Logical function symbol. No side effects, cannot be used in non-ghost program code.

`predicate`

Logical predicate symbol. Special case of `function` with Boolean return type.

`let function`, `val function`, `let predicate`, `val predicate`

Program function with no side effects, can be used in both programs and logic.

`let lemma`

Ghost program function (without side effects). The specification is added to the logical context as a lemma which states "for all values of the parameters, the precondition of this function implies the postcondition".

In addition, all program functions may be declared `ghost` (e.g., `let ghost function ...`). Ghost functions may only be called in ghost code and must have no side effects.

2.3 Computing the verification conditions

Let us provide some more details on how verification conditions are computed. This section first goes over the weakest-precondition calculus used by Why3, and then describes how it interacts with aliasing.

2.3.1 Weakest precondition calculus

WhyML functions are annotated with contracts based on Hoare logic [51, 40]. Given an expression e, a precondition P and a postcondition Q, where P and Q are assertions in first-order logic, one can write the *Hoare triple* $\{P\}e\{Q\}$. Its intuitive reading is: Whenever the state before the execution of e is such that P holds, then the computation terminates, there is no runtime error, and the final state satisfies Q. Hoare triples can be derived from a set of inference rules based on program syntax. A soundness theorem implies that any derivable triple is correct. The inference rule for the let-in construct is shown in Figure 2.1.

$$\frac{\{P\}e_1\{Q[v \leftarrow \texttt{result}]\} \quad \{Q\}e_2\{R\}}{\{P\}\texttt{let } v = e_1 \texttt{ in } e_2\{R\}}$$

Figure 2.1: Some Hoare triple inference rules.

The contract of a function forms a Hoare triple with the function body. The precondition P is the conjunction of the `requires` clauses, and the postcondition Q is the conjunction of the `ensures` clauses. For example, consider again the ancient Egyptian multiplication algorithm from the previous section. It is reproduced in Fig. 2.2 below.

Using this contract, Why3 produces a logical goal that implies that the program satisfies this specification. The aim is to find out whether the Hoare triple defined by the contract is valid. Using the inference rules naively would be far too tedious. Indeed, the rule for sequences of instructions (see Figure 2.1) requires an intermediate assertion between each pair of statements. The code would need to be very heavily annotated for the naive approach to work.

Instead, Why3 uses Dijkstra's weakest-precondition calculus [28]. We define the predicate transformer $\mathrm{WP}(.,.)$. Where e is an expression and Q a postcondition, $\mathrm{WP}(e, Q)$ computes the weakest precondition P such that $\{P\}e\{Q\}$ holds. This is a way to automate the search for intermediate assertions in the derivations. A subset of the computation rules for $\mathrm{WP}(.,.)$ can be found in Figure 2.3.

The soundness theorem of WP states that for any e and Q, the triple $\{\mathrm{WP}(e, Q)\}e\{Q\}$ is valid. Therefore, to show that a contract $\{P\}e\{Q\}$ is valid,

```
let mul (a b: int) : int
  requires { b ≥ 0 ∧ a ≥ 0 }
  ensures  { result = a * b }
=
  let ref x = a in
  let ref y = b in
  let ref z = 0 in
  while y ≠ 0 do
    invariant { y ≥ 0 }
    invariant { z + x * y = a * b }
    variant   { y }
    if mod y 2 = 1 then z ← z + x;
    x ← 2 * x;
    y ← div y 2
  done;
  z
```

Figure 2.2: Ancient Egyptian multiplication.

$$\mathrm{WP}(t, Q) = Q[\texttt{result} \leftarrow t]$$

$$\mathrm{WP}(x \leftarrow t, Q) = Q[x \leftarrow t]$$

$$\mathrm{WP}(e_1; e_2, Q) = \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q))$$

$$\mathrm{WP}(\texttt{let}\ \ v = e_1\ \ \texttt{in}\ \ e_2, Q) = \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[v \leftarrow \texttt{result}])$$

$$\mathrm{WP}(\texttt{assert}\ \ \{R\}, Q) = R \land (R \rightarrow Q)$$

Figure 2.3: Some WP rules.

it suffices to prove that $P \rightarrow \mathrm{WP}(e, Q)$. Why3 computes $\mathrm{WP}(e, Q)$ and outputs $P \rightarrow \mathrm{WP}(e, Q)$ as a logical goal for the user to prove.

Why3 requires annotating loops with an invariant. The corresponding WP computation makes sure that this invariant is valid at the start of the loop, maintained through one iteration of the loop, and that at the end of the loop, it implies the required postcondition. A simplified version of this rule is shown in Figure 2.4. It assumes that the only modification of the memory state in the loop occurs on a variable v, and does not handle termination.

$$\begin{array}{lr}
\mathrm{WP}(\texttt{while}\ \ t\ \ \texttt{do}\ \ \texttt{invariant}\ \ \{J\}\ e\ \ \texttt{done}, Q) \equiv & \\
\quad J \land & [\text{invariant initialization}] \\
\quad \forall v. & \\
\qquad (J \land t \rightarrow \mathrm{WP}(e, J)) \land & [\text{invariant preservation}] \\
\qquad (J \land \neg t \rightarrow Q) & [\text{loop exit and postcondition}]
\end{array}$$

Figure 2.4: Simplified WP rule for `while`.

Function calls, modularity

Much like loops, function calls are handled in a modular way in the computation of verification conditions. When computing WP(`f(x)`, Q), Why3 does not look inside the body of `f`, only its specification is used. This is why abstract `val` functions can be called in just the same way as regular functions. A simplified version of the WP rule for function calls can be found in Fig. 2.5. It assumes that the function f has a single formal parameter x, and that it does not write into any mutable region that is not a local variable of f. The formulas P_f and Q_f denote the precondition and postcondition of f, respectively.

$$\begin{aligned}\mathrm{WP}(f\ t, Q) \equiv{}& P_f[x \leftarrow t] \wedge && \text{[precondition of } f\text{]}\\ & (\forall \texttt{result}.\ Q_f[x \leftarrow t]) \rightarrow Q && \text{[postcondition of } f \text{ implies } Q\text{]}\end{aligned}$$

Figure 2.5: Simplified WP rule for function calls.

Although it forces users to write strong enough specifications for their functions, hiding the details of the functions that are called from the proof context is a useful feature. Indeed, automated solvers would be much less efficient if the context was polluted by useless details about the implementation of called functions. This problem may still occur when verifying very large functions (on the order of hundreds of lines). By the end of the function body, the proof context contains all sorts of details about the previous instructions. Some of these details are needed, but some are not. This makes automated solvers quite slow. One solution is to cut large functions into many smaller ones to take advantage of proof modularity, but it is not always practical.

In order to mitigate this problem, WhyML expressions may be isolated into abstract blocks (also called "black boxes"). If `e` is a valid WhyML expression, it can be given a specification using the syntax `begin ensures { f } e end` (preconditions are allowed but typically not needed). A verification condition is generated for the postcondition `f` of `e`, and in the rest of the program, the details of `e` are hidden from the proof context, only its specification remains. For verification concerns, this is exactly the same as if `e` was enclosed in a function call. However, the user does not need to write a precondition, and the structure of the program does not need to be changed artificially for the sake of proof convenience.

2.3.2 Aliasing

It is relatively easy to build WhyML programs that involve aliases, that is, multiple separate ways to access the same memory zone. Consider the example from Fig. 2.6. The function `double_incr` increments two integer references passed as parameters. What if they are the same?

The WP rules from Sec. 2.3 break down in presence of aliasing. For example, the rule for assignment is $\mathrm{WP}(x \leftarrow t, Q) = Q[x \leftarrow t]$. If the variables x and y are aliased, then the correct answer would instead be $Q[x \leftarrow t, y \leftarrow t]$. In short, in order to compute $\mathrm{WP}(x \leftarrow t, Q)$, we need to statically know which variables are aliased to x. In general, all ways to access a given memory location must

```
1  let double_incr (ref x y: int) =
2    ensures { x = old x + 1 ∧ y = old y + 1 }
3  =
4    x ← x + 1;
5    y ← y + 1
6
7  let error () =
8    let ref x = 0 in
9    double_incr x x;
10   assert { x = 1 } (* true from the postcondition. But x=2... *)
```

Figure 2.6: An ill-typed WhyML program.

be statically known.

Alias tracking is performed through the type system. Mutable types carry an identity token called *region* [44]. For example, an integer reference might have type `ref int` ρ. Two variables are aliased if and only if they have the same region. Some programs do not type check, such as `if c then x else y` if `x` and `y` have regions that cannot be unified. Indeed, Why3 would be hard-pressed to infer the region carried by the type of this expression. Conversely, formal parameters of functions are assumed to be separated. This resolves the troubling example of `double_incr` in a simple way. The call to `double_incr` at line 9 is ill-typed, so Why3 rejects it.

This static alias tracking does not come for free. For example, we cannot write functions whose parameters are maybe-aliased. Neither can we declare recursive mutable types, such as a mutable linked list. Indeed, they might carry an unbounded number of regions.

Resets

Some expressions change the aliasing status of mutable objects. This commonly occurs with double references. Consider the following type of resizable arrays. It consists of a record with a mutable array field.

```
type array 'a =
  private { mutable ghost elts : int → 'a;
            length : int }
  invariant { 0 ≤ length }

val make (n: int) (v: 'a) : array 'a
  requires { n ≥ 0 }
  ensures { forall i:int. 0 ≤ i < n → result[i] = v }
  ensures { length result = n }

type r = { mutable data: array int }
```

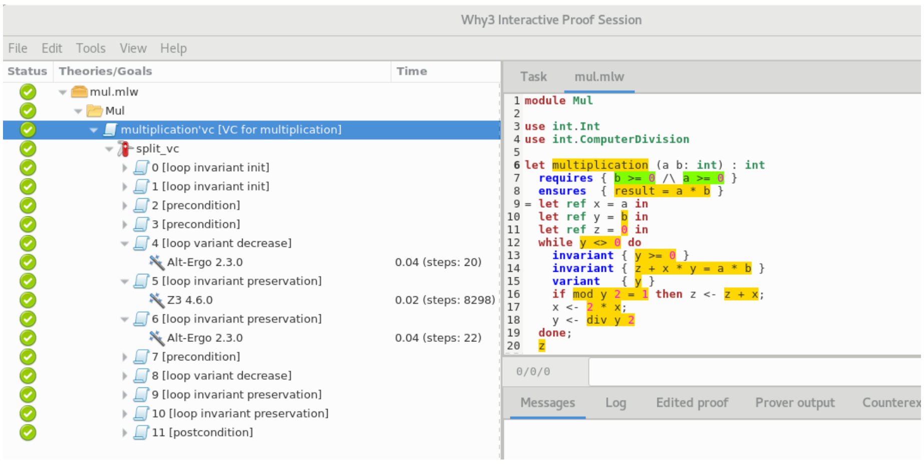
A variable of type `r` can be resized by substituting its `data` field with a new array. However, this breaks the region-alias correspondence:

```
  let x = { data = Array.make 10 0 } in
  let olddata = x.data in
  (* x : r ρ (array int ρ1) *)
  let newdata = Array.make 20 0 in
  (* newdata: array int ρ2*)
  x.data ← newdata
```

The assignment at line 6 can be seen as attempting to change the type of `x` from `r` ρ (`array int` ρ_1) to `r` ρ (`array int` ρ_2). However, we cannot simply update the type. For example, what about the expression "`if ... then x.data <- newdata else ()`"? Rejecting these programs outright would be too restrictive. However, the only way to observe the type inconsistency is through `olddata` and `newdata`, which are the only remaining ways to access ρ_1 or ρ_2 outside `x` itself. Therefore, Why3 accepts the program and invalidates the variables `olddata` and `newdata`. More precisely, it *resets* the regions ρ_1 and ρ_2 so that the only way to access them is through `x`, so it does not matter if `x` contains ρ_1 or ρ_2 in terms of aliases with the rest of the program. So, the program above is accepted, but if we attempt to use `olddata` or `newdata` later, Why3 raises a type error stating that the assignment at line 6 prevents further use of the variable.

2.4 Why3 in practice

Assume we have written a well-formed WhyML file. How do we verify it practice? The usual way to do so is through Why3's graphical interface (Fig. 2.7). Why3 computes the verification conditions of each function and splits them into independent subgoals. It is then up to the user to discharge them. Why3 interfaces with various SMT solvers by outputting the verification conditions in their native syntax. For example, in the screenshot from Fig. 2.7, goals 4 and 6 have been verified using Alt-Ergo, and goal 5 using Z3. Interactive theorem provers such as Coq may also be used.



```
mul.mlw
  Mul
    multiplication'vc [VC for multiplication]
      split_vc
        0 [loop invariant init]
        1 [loop invariant init]
        2 [precondition]
        3 [precondition]
        4 [loop variant decrease]
          Alt-Ergo 2.3.0                          0.04 (steps: 20)
        5 [loop invariant preservation]
          Z3 4.6.0                                0.02 (steps: 8298)
        6 [loop invariant preservation]
          Alt-Ergo 2.3.0                          0.04 (steps: 22)
        7 [precondition]
        8 [loop variant decrease]
        9 [loop invariant preservation]
        10 [loop invariant preservation]
        11 [postcondition]
```

```
module Mul

use int.Int
use int.ComputerDivision

let multiplication (a b: int) : int
  requires { b >= 0 /\ a >= 0 }
  ensures  { result = a * b }
= let ref x = a in
  let ref y = b in
  let ref z = 0 in
  while y <> 0 do
    invariant { y >= 0 }
    invariant { z + x * y = a * b }
    variant   { y }
    if mod y 2 = 1 then z <- z + x;
    x <- 2 * x;
    y <- div y 2
  done;
  z
```

0/0/0

Figure 2.7: Why3's graphical interface.

Another question is the purpose of WhyML programs. Given the ability to verify their correctness, one may wonder how to leverage it to verify programs written in more widely-used programming languages.

One way to do so is translating programs from other languages into WhyML. Through various front-ends (Frama-C/WP, Krakatoa, GNAT), WhyML can be used as an intermediate language for the verification of C, Java, or Ada programs [36, 58, 22].

Another possibility is to go the other way around. Why3's extraction mechanism can compile verified WhyML programs into correct-by-construction OCaml programs. One contribution of this work is the capability to do the same for C programs. This is the main focus of the next chapter.

Chapter 3

Verifying C programs with Why3

This chapter presents two contributions. The first is a model of the C language in WhyML. It includes an axiomatization of C's memory management, as well as various data types. The second contribution is an extraction mechanism from WhyML to C. Using these, Why3 can now be used to verify C programs through extraction. The workflow we advocate is as follows. First, we reimplement the C program we want to verify as a WhyML program. The WhyML model of the C language allows WhyML programs to be written in a style that is very close to C, so this translation from C to WhyML is often straightforward. Then, we verify the WhyML program in any of the usual ways. Finally, using Why3's extraction mechanism, we produce verified C code. In this chapter, we walk through this workflow and verify a tiny C library that implements a ring buffer data structure. The chapter is structured as follows. We start by presenting our models of C integer and character types (Sec. 3.1). In Sec. 3.2, we present a ring buffer library from the open-source operating system Contiki. We go over its C source code, a transcription in WhyML, and a proof of the latter's correctness. Section 3.3 presents our model of C's memory. We outline the extraction mechanism from WhyML to C in Sec. 3.4. Finally, in Sec. 3.5, we review related work and evaluate our design choices.

3.1 Modeling C types

Before being able to write C programs in WhyML, we must ensure that the semantics of the various C types are preserved. For example, WhyML's built-in `int` type corresponds to mathematical integers, with no consideration of overflow. Therefore, C programs that involve machine integer types such as `uint64_t` cannot be written faithfully using WhyML's `int` type. Instead, we need to use some other Why3 types. This section presents axiomatics for the various C integer and character types. Let us start by outlining the correctness requirements for our various models.

Broadly speaking, a Why3 model of a C integer type, say `uint64_t`, consists in a theory that declares a corresponding type and all the relevant functions. The functions are typically not given a body. Instead, they are only given a specification. From a logical standpoint, we are adding extra axioms to our context. Let us now explain what we expect of them with an example. The following snippet is an incomplete sketch of what could be a model of the uint64_t type. A more complete model can be found in Sec. 3.1.1.

```
type uint64 = < range 0 0xffffffffffffffff >

val add (x y:uint64) : uint64
  ensures { uint64'int result
          = mod (uint64'int x + uint64'int y) (uint64'maxInt + 1) }
```

The range constructs defines a type that ranges over an interval of integers. It also adds the symbols uint64'int, uint64'minInt and uint64'maxInt to the context. The uint64'int function maps uint64 values to int values between 0 and $2^{64} - 1$. It is a logical function, but not a program function, so it can only be used in specifications and assertions. The uint64'maxInt constant is the largest element of the uint64 type. More details on range types can be found in Sec. 2.1.3.

Conversely, the add function is a program function. The val keyword means that the function does not have a body, and that its specification is an axiom rather than a theorem.

When adding new axioms to a logical system, the first thing to ensure is that the system remains logically consistent. In our case, there is more to do. The goal is not only to prove a WhyML program, but also the corresponding extracted C code. However, the add function does not have a body to compile to C. So, calls to add need to be translated to some C code somehow. This is done using a driver file. For each WhyML type and val declaration, the driver needs to contain a directive that explains how to translate it to C. The driver directives are as much a part of the model as the WhyML theory. In our case, the relevant driver directives might look like the following.

```
syntax type uint64 "uint64_t"
syntax val add    "%1 + %2"
```

The meaning of the directive for add is that any call to add in the WhyML source code is replaced by the string template, substituting the first argument to add for "%1" and the second argument for "%2". The A-normalization performed by Why3 ensures that the arguments are plain variables, so there is no need to worry about the side effects of the computations of the arguments.

In addition of the logical consistency of the model of uint64_t, we also need to ensure the following. First, any valid WhyML program should be either extracted to a valid C program (syntactically correct and free of undefined behavior), or rejected by the extraction. Second, the extracted code should fulfill the specification of the WhyML program. In the case of uint64_t, the meaning of this is somewhat clear. The abstract functions of the module have preconditions on the function parameters and postconditions that talk only about the returned value. The C standard [53] specifies what the result of the corresponding C operators, or combinations of operators should be. The model is correct if for all inputs that validate the precondition, the extracted code does not invoke undefined behavior and the result validates the postcondition.

However, there are more complex cases. The specifications may refer to things that are not mirrored in the extracted C code, such as ghost fields. In these cases, the question of the consistency of the model is not yet clearly answered. Establishing a formal semantics of WhyML and a proof of correctness of the extraction is a significant challenge left for future work.

3.1.1 Integer types

Why3's standard library contains a modular axiomatization of machine integers. The `Bounded_int` theory contains axioms and functions symbols for integers that take values between `min` and `max`, which are left undefined. The `Unsigned` theory is a specialized version of the `Bounded_int` theory that specifies that `min` is 0. These theories can then be instantiated with appropriates values of the bounds to model integer types of various lengths. Figure 3.1 presents abridged versions of these theories. Notice the `meta coercion` declaration. When it is present, the `to_int` function is automatically applied in logical assertions and specifications. Otherwise, the postcondition of `(+)` would contain three occurrences of `to_int`.

In addition to `Int32`, the standard library contains instances of `Bounded_int` for unsigned 32-bit integers, as well as (signed and unsigned) 64-bit integers, and 63-bit integers for OCaml. Users may easily generate new instances with the `clone` directive to model integers of different sizes.

Let us now focus on the arithmetic primitives more closely. The `Bounded_int` `(*)` primitive implements what we could call defensive multiplication; it requires, as a precondition, that the product of the operands does not overflow. It is the only legal way to compute multiplications on C signed integers. Indeed, multiplying two signed integers when their product is not in the range of representable integers invokes undefined behavior. However, computations on C unsigned integers are allowed to overflow, in which case the results silently wrap. Therefore, the theory of unsigned integers features extra multiplication primitives, in addition to the `(*)` operator inherited from `Bounded_int`. These operators are shown in Figure 3.2. The function `mul_mod` computes the product of two unsigned integers with wraparound semantics. At extraction, both `(*)` and `mul_mod` can safely be replaced by the C operator `(*)` (see Sec. 3.4.4). In WhyML programs, it is up to the user to choose between the two depending on the proof context. Evidently, the postcondition of `(*)` is stronger, but its precondition is stronger as well. Finally, `mul_double` computes the product of two integers and returns the full product as a pair of integers. This primitive does not correspond to a standard C operator, but can be implemented using inline assembly or other compiler-specific features (see Sec. 4.8.2 for an in-depth discussion). Similarly to multiplication, addition and subtraction also come in different flavors (defensive against overflow, two's complement, with carry in/out).

Note that the translation from `mul_mod` to the C operator `*` assumes that the type `t` is at least as wide as the C `int` type, or the operands would be implicitly converted to `int` first through integer promotion. (See Sec. 3.2.2 for an explanation of integer promotion and an example.) As a result, our models of 32-bit integers implicitly assume that the C `int` type has bit-width 32 or less. In the end, we will assume that it is exactly 32 bits wide.

```
module Bounded_int

  type t

  constant min : int
  constant max : int

  function to_int (n:t) : int
  meta coercion function to_int (* Cast t to int in logical declarations *)

  val to_int (n:t) : int
    ensures { result = n }

  predicate in_bounds (n:int) = min ≤ n ≤ max

  axiom to_int_in_bounds: forall n:t. in_bounds n

  val ( + ) (a:t) (b:t) : t
    requires { [@expl:integer overflow] in_bounds (a + b) }
    ensures  { result = a + b }

  val ( * ) (a:t) (b:t) : t
    requires { [@expl:integer overflow] in_bounds (a * b) }
    ensures  { result = a * b }

end

module Unsigned

  let constant min_unsigned : int = 0

  clone export Bounded_int with constant min = min_unsigned, axiom .

  constant radix : int = max + 1

end

module Int32

  type int32 = < range -0x80000000 0x7fffffff >

  clone export Bounded_int with
    type t = int32,
    constant min = int32’minInt,
    constant max = int32’maxInt,
    function to_int = int32’int

end
```

Figure 3.1: Bounded integers in Why3.

```
val mul_mod (x y:t) : t
  ensures { to_int result = mod (to_int x * to_int y) radix }

val mul_double (x y:t) : (t,t)
  returns { (r,q) →
    to_int r + radix * to_int q =
    to_int x * to_int y }
```

Figure 3.2: Unsigned multiplication primitives.

3.1.2 Characters, strings

Strings are a built-in type in Why3, with no underlying notion of characters. The main operations are string concatenation and substring extraction. There is a notion of length, but it is axiomatized rather than based on a number of characters. One of the main goals of this choice is to adopt the same modelization of strings as the most popular SMT solvers that support them. Another reason is to allow support for multiple sets of characters (ASCII, Unicode, etc.) rather than choosing a specific character set to base the modelization upon. The Why3 standard library also contains a theory of 8-bit characters. The characters are identified with strings of length 1, and are mapped to integers from 0 to 255. This character type is intended to be mapped to OCaml's `char` type. Figure 3.3 shows a short excerpt of this theory.

```
type char = abstract {
  contents: string;
} invariant {
  length contents = 1
}

axiom char_eq: forall c1 c2.
  c1.contents = c2.contents → c1 = c2

function code char : int

axiom code: forall c. 0 ≤ code c < 256
```

Figure 3.3: Why3's `char` type.

Let us now discuss how to build a suitable model of C strings and characters upon this. There are three character types in C: `char`, `signed char`, and `unsigned char`. Let us first review the assumptions we can make about them. The C standard imposes that characters must have bit length at least 8 (minimum value of `CHAR_BIT`). We will make the assumption that their length is exactly 8. From there, we can easily model `signed char` and `unsigned char` using range types (`schar` and `uchar`) covering the intervals $[-128, 127]$ and $[0, 255]$ respectively. As instances of `Bounded_int`, they inherit the same arithmetic primitives as the integer types. The situation is a bit more complex for `char`. The C standard states that `char` shall have the same range, representation and behavior as one of the other two character types, but leaves it up to the implementation to decide which one. For the sake of portability, we would rather not make that choice. Rather, we make the assumption that the implementation uses the seven-bit ASCII character set. Whether characters are signed or unsigned, we assume that the characters we care about the most are in the range $[0, 127]$, which fits both `signed char` and `unsigned char`. Thus, we can underspecify the C `char` type. We represent it using the type of Why3 ASCII characters, so it does not get any arithmetic primitives. The `code` function that converts from `char` to `int` maps characters to the interval $[0, 255]$, but this does not mean we assume that characters are unsigned. Indeed, the only place where the value of `code` matters is if it used to convert to some other type which allows arithmetic. We define such conversion functions between `char` and `uchar`, as well as between `char` and `schar`. However, we only specify the behavior of these functions for characters between 0 and 127. Figure 3.4 contains an excerpt of the theory of `unsigned char`. The `signed char` theory is analogous.

```
type uchar = < range 0 255 >

let constant max_uchar : int = 255
function to_int (x:uchar) : int = uchar’int x
let constant length : int = 8
let constant radix : int = 256

(* unsigned integer arithmetic *)
clone export mach.int.Unsigned with
  type t = uchar,
  constant max = uchar’maxInt,
  constant radix = radix,
  goal radix_def,
  function to_int = uchar’int

(* char can be either signed or unsigned *)
val function of_char (x:char) : uchar
  ensures { 0 ≤ code x ≤ 127 → result = code x }

val function to_char (x:uchar) : char
  ensures { 0 ≤ x ≤ 127 → code result = x }
```

Figure 3.4: `unsigned char` in Why3.

Now that we have a model for C characters, we can discuss strings. C strings are simply null-terminated arrays of characters, so we do not need to write WhyML programs that use the `string` type. We can simply use pointers to the various kinds of characters. The only exception to this is string literals. In WhyML programs, string literals are interpreted as values of the `string` type, and we need some way to allow these in programs destined to be extracted to C.

For now, we only support string literals in a limited way. The `string` type is extracted to the C type `char*`. WhyML programs meant to be extracted to C can read from values of the `string` type using the following `get` function, but not write into them.

```
val get (s:string) (i:int32) : char
  requires { 0 ≤ i ≤ length s }
  ensures  { i < length s → result = s[i] }
  ensures  { i = length s → result = chr 0 }
```

Let us explain the specification of `get`. In WhyML's model of characters, string literals work the same way as OCaml strings. The `([])` operator is a logical function that has the same semantics as the OCaml function `String.get` as long as the argument is between 0 (included) and the length of the string (excluded). For example, in the program `let s = "abc" in ...`, then `length s` is 3, `s[0]` is the character `a`, and `s[2]` is the character `c`. The specification of `([])` does not define the values `s[3]` and `s[-1]`. In C, string literals work in a similar way, with one key difference. In the program `char* s = "abc"`, `s[3]` is the null character. This is taken care of in the second postcondition of `get` above. The precondition ensures that we can only read inside the string or one past its end (dereferencing `s[-1]` or `s[4]` would invoke undefined behavior).

This function is enough to use read-only string literals in WhyML programs that otherwise manipulate strings as pointers to characters. An example can be found in Sec. 4.7.

3.2 Example: Contiki's ring buffer library

A ring buffer (or cyclic buffer) is a data structure that uses a buffer as if its end and its beginning were connected, that is, as if it was circular. It is well-suited to implement a FIFO queue with a maximum length, as both reads and writes are constant time with no need to shift the elements around when the end of the buffer is reached.

Contiki is an open-source operating system designed for embedded systems and IoT devices with small amounts of memory. It is written in C. As part of its standard library, it contains a ring buffer implementation. Let us review this C program and attempt to produce a verified version using Why3.

3.2.1 C code overview

The following listings are taken from the `ringbuf.h` header and the `ringbuf.c` file in the sources of Contiki 2.6. Comments have been edited for readability.

```c
/**
 * Structure that holds the state of a ring buffer. This
 * struct is an opaque structure with no user-visible
 * elements.
 */
struct ringbuf {
  uint8_t *data;
  uint8_t mask;
  uint8_t put_ptr, get_ptr;
};
```

Figure 3.5: Ring buffer definition.

The `ringbuf` struct is defined in Figure 3.5. It contains a pointer to its data (an array of bytes), as well as the offsets `put_ptr` and `get_ptr`. `get_ptr` is the offset of the oldest element in the buffer (that is, the next one to be read). `put_ptr` points one past the newest element in the buffer, that is, the offset where the next element should be written. Finally, `mask` stores information about the size of the buffer. More specifically, the size of the buffer is always a power of 2, and `mask` is the size of the buffer minus one. This allows for fast computations modulo the buffer size. Indeed, for all non-negative `x`, we have x % (mask + 1) == x & mask, because the binary representation of mask is a block of all zeroes followed by a block of all ones. The ring buffer is initialized by providing an external buffer to store the data (Figure 3.6).

```
void
ringbuf_init(struct ringbuf *r, uint8_t *dataptr, uint8_t size)
{
  r->data = dataptr;
  r->mask = size - 1;
  r->put_ptr = 0;
  r->get_ptr = 0;
}
```

Figure 3.6: Ring buffer initialization.

Let us now discuss the invariants of the ring buffer structure more precisely. The ring buffer contains up to size - 1 elements, where size is the length of the data array and is a power of two. The number of elements currently stored in the buffer is equal to the difference put_ptr - get_ptr (all offset computations are modulo size). The elements are stored in the range from offset get_ptr (for the oldest element) to put_ptr - 1 (for the newest).

A couple of small utility functions, reproduced in Figure 3.7, make the length of the buffer and the number of stored elements accessible to the outside world (the actual fields of the structure are private).

```
int ringbuf_size(struct ringbuf *r)
{
  return r->mask + 1;
}

int ringbuf_elements(struct ringbuf *r)
{
  return (r->put_ptr - r->get_ptr) & r->mask;
}
```

Figure 3.7: Ring buffer utility functions.

The elements are stored and read in a first-in-first-out order (FIFO), as shown in Figure 3.8. The ringbuf_put function checks if the buffer is full (by comparing the number of elements currently stored against the maximum amount). If it is not full, it stores the new element at offset put_ptr and increments that offset. Similarly, the ringbuf_get function checks that the buffer is not empty, and if it is not, it reads the element at read_ptr and increments the offset.

3.2.2 Contiki's ring buffer, in WhyML

Let us now try to implement Contiki's ring buffer in WhyML. The objective is to do so in a way that preserves the semantics of Contiki's ring buffer and that can be extracted to a similar C program.

```
int
ringbuf_put(struct ringbuf *r, uint8_t c)
{
  /* Check if buffer is full. If it is full, return 0 to indicate that
     the element was not inserted into the buffer.
  */
  if(((r->put_ptr - r->get_ptr) & r->mask) == r->mask) {
    return 0;
  }
  r->data[r->put_ptr] = c;
  r->put_ptr = (r->put_ptr + 1) & r->mask;
  return 1;
}

int
ringbuf_get(struct ringbuf *r)
{
  uint8_t c;

  /* Check if there are bytes in the buffer. If so, we return the
     first one and increase the pointer. If there are no bytes left, we
     return -1.
  */
  if(((r->put_ptr - r->get_ptr) & r->mask) > 0) {
    c = r->data[r->get_ptr];
    r->get_ptr = (r->get_ptr + 1) & r->mask;
    return c;
  } else {
    return -1;
  }
}
```

Figure 3.8: Ring buffer write and read.

```
type uint8 = < range 0 0xff >
let constant max_uint8 : int = 0xff
let constant radix : int = max_uint8 + 1
let constant zero_unsigned : uint8 = 0

clone export mach.int.Unsigned with
  type t = uint8,
  constant max = uint8’maxInt,
  constant radix = radix,
  constant zero_unsigned,
  function to_int = uint8’int

let add’ (x y:uint8) : int32
  ensures { result = x + y } (* no modulo *)
= Int32.(+) (to_int32 x) (to_int32 y)

let sub’ (x y:uint8) : int32
  ensures { result = x - y } (* no modulo *)
= Int32.(-) (to_int32 x) (to_int32 y)

val band_pow2 (x:int32) (y:uint8) (ghost exp:int) : int32
  requires { y = power 2 exp - 1}
  ensures  { result = mod x (power 2 exp) }
```

Figure 3.9: The `uint8` type and its arithmetic operations.

Integers, arithmetic primitives. The `ringbuf` C struct contains four fields, one of type `uint8_t*` and three of type `uint8_t`. Therefore, the first thing to do is to model the `uint8_t` type and its arithmetic operations. The model can be found in Fig. 3.9.

We model unsigned 8-bit integers as an instance of bounded integers using the `clone` construct. Unfortunately, the operations introduced by `clone` are not faithful translations of their C counterparts. Indeed, if `x` and `y` are two C variables of type `uint8_t`, the C expression `x + y` does not compute an `uint8_t` that is the sum of `x` and `y` modulo 2^8. Instead, since all values of type `uint8_t` can be represented by an `int`, `x` and `y` are converted to the type `int`. The result is the sum of `x` and `y`, even if it is greater than 2^8. (It cannot be greater than the maximum integer, as the standard guarantees that it is at least $2^{16} - 1$.) This implicit conversion is called *integer promotion* in the C standard [53, 6.3.1.1].

In order to model arithmetic operations on the `uint8_t` type in a way that takes integer promotion into account without littering the WhyML code with explicit type conversions, we define extra arithmetic primitives `add’` and `sub’`. (In principle, we would need to do the same for other operations such as multiplication, but we do not need them for this example.) We make the assumption that the C type `int` is exactly 32 bits wide, so `int` and `int32_t` are the same type. Both operations take two operands of type `uint8`, convert them to `int32`, and compute a result of type `int32`. They model the C operators `+` and `-` on the `uint8_t` type faithfully, and can be replaced by them at extraction. They do not require a precondition stating the absence of overflow, as explained in the previous paragraph.

In addition to the more usual arithmetic operations, Contiki’s ring buffer makes heavy use of the bitwise `&` operator. It is not present in the `Unsigned` module, so we need to axiomatize it. Our model of machine integers is not well suited to model the full semantics of a bitwise operator, compared to e.g., a model based on bit vectors [42]. However, the only way the `&` operator is used in the program is to compute modulo the buffer size. Therefore, we can axiomatize an underspecified, restricted version.

The `band_pow2` function has the same behavior as the `&` operator provided that the second operand is a power of 2 minus one, that is, it reduces the first operand modulo the power of 2. A third, ghost argument states the exponent. Much like the other arithmetic operations, this function is subject to integer promotion, and returns an `int32`. However, it is always used in contexts where the first argument is already an `int32`, because it is the result of an addition or subtraction. Therefore, in order to decrease the number of explicit casts in the WhyML code, the first argument has type `int32`. At extraction, the third argument is erased and `band_pow2` can safely be replaced by C's `&` operator.

The `ringbuf` type. Let us now discuss the ring buffer implementation proper. We will save the discussion on how to model pointers for later (Sec. 3.3), and assume that the type `ptr 'a` is a polymorphic pointer type. We can now define the `ringbuf` type rather straightforwardly.

```
type ringbuf = { mutable data   : ptr uint8;
                 mutable mask   : uint8;
                 mutable put_ptr: uint8;
                 mutable get_ptr: uint8; }
```

Recall the invariants on the ring buffer data structure. Its length should be a power of 2, `mask` should be the length minus 1. Both offsets should lie between 0 and `mask`. The contents of the buffer are the elements from `get_ptr` to `put_ptr - 1`, in that order. These invariants can be expressed as type invariants of the `ringbuf` record. Using *ghost* fields, we can express abstract concepts such as the contents of the buffer without changing the semantics of the program.

```
type ringbuf = {
  mutable data: ptr uint8; mutable mask: uint8;
  mutable put_ptr: uint8; mutable get_ptr: uint8;
  ghost mutable size: int; ghost mutable exp: int;
  ghost mutable contents: Seq.seq uint8;
}
  (* constraints on buffer size *)
  invariant { 1 ≤ size ≤ uint8'maxInt }
  invariant { 0 ≤ exp }
  invariant { size = power 2 exp }
  (* field sanity checks *)
  invariant { mask = size - 1 }
  invariant { 0 ≤ put_ptr < size }
  invariant { 0 ≤ get_ptr < size }
  (* data pointer validity *)
  invariant { valid data size }
  invariant { writable data }
  (* semantics of the buffer *)
  invariant { Seq.length contents = mod (put_ptr - get_ptr) size }
  invariant { forall i. 0 ≤ i < Seq.length contents →
              Seq.get contents i = data[mod (get_ptr + i) size] }
  (* instance of valid buffer *)
  by { data = dummy_nonnull (); mask = 0;
       put_ptr = 0; get_ptr = 0;
       size = 1; exp = 0; contents = Seq.empty; }
```

Let us examine the ghost fields and invariants more closely. The `size` field is there mostly for convenience, we could simply write `mask + 1` everywhere instead. The `exp` field is there to express the invariant that `size` is a power of two. It is not the only way to go about it. Instead, we could write the relevant invariant with an existential quantifier. However, it would make the proofs a little more complex, as automated provers are not very good at proving existential formulas. Moreover, it can be useful to have access to the actual power of two, rather than just knowing that it exists. This will be particularly relevant when specifying the behavior of the bitwise and operator. The invariants on `data` state that the data buffer is valid and that we are allowed to write inside it (more details on the semantics of `valid` and `writable` can be found in Sec. 3.3). The `contents` field stores the sequence of elements currently stored in the buffer. The corresponding type invariants specify the number of elements currently in the buffer, as well as their position in the buffer. Making this explicit allows us to express the specifications of the functions that read and write into the buffer in a natural way later on.

Finally, using the `by` connective, we exhibit an example of a record that satisfies all the invariants. Indeed, Why3 only accepts the definition of a type with invariants on the condition that the type is not empty. The simplest way to prove this is to provide a valid element of that type. The `dummy_nonnull` function is a `val` declaration. It corresponds to an axiom that states that a non-null pointer of type `uint8_t` exists.

Ring buffer operations. Let us now implement and prove the rest of the ring buffer library. The initialization function can be implemented straightforwardly. Its code is identical to the C version, except that we also have to initialize the ghost fields. We defer the initialization of the `exp` field to the caller by taking the exponent as an extra ghost parameter. The proof mostly consists in checking that the type invariants of `ringbuf` are upheld, but this is an easy task, mostly because `r.contents` is empty.

```
let ringbuf_init
    (ref r: ringbuf) (dataptr: ptr uint8) (size: uint8) (ghost exp:int)
  requires { valid dataptr size }
  requires { writable dataptr }
  requires { 0 ≤ exp }
  requires { size = power 2 exp }
  requires { 1 ≤ size }
  ensures  { r.size = size ∧ r.put_ptr = 0
             ∧ r.get_ptr = 0 ∧ r.contents = Seq.empty }
=
  r.data ← dataptr;
  r.mask ← size - 1;
  r.put_ptr ← 0;
  r.get_ptr ← 0;
  r.size ← uint8'int size;
  r.exp ← exp;
  r.contents ← Seq.empty
```

The implementation and proof of the utility functions is similarly straightforward. Note the `ref` keyword before the ring buffer function arguments. It signifies that a pointer to the ringbuffer is passed to the functions (no copy occurs). At extraction, this is translated to an argument of type `struct ringbuf*`, rather than `struct ringbuf`.

```
let ringbuf_size (ref r:ringbuf) : int32
  ensures { result = r.size }
= add' r.mask 1

let ringbuf_elements (ref r:ringbuf) : int32
  ensures { result = Seq.length r.contents }
=
  band_pow2 (sub' r.put_ptr r.get_ptr) r.mask r.exp
```

The proof of `ringbuf_put` and `ringbuf_get` is more difficult. Recall the specification of `ringbuf_put` (see Fig. 3.8). It attempts to write an element into the ring buffer. It returns either 0, in which case the buffer was full and nothing was written, or 1, in which case the write was successful. The difficulty does not lie in proving this specification, but in ensuring that the invariants of the ring buffer structure were preserved. The WhyML specification of `ringbuf_put` is as follows. Note that as explained earlier, we have assumed that the C types `int` and `int32_t` are the same, so we use `int32` as return type.

```
let ringbuf_put (ref r: ringbuf) (c: uint8) : int32
  ensures { 0 ≤ result ≤ 1 }
  ensures { r.put_ptr = mod (old r.put_ptr + result) r.size }
  ensures { Seq.length (old r.contents) < r.size - 1 ↔ result = 1 }
  ensures { result = 0 → r.contents = old r.contents }
  ensures { result = 1 → r.contents = Seq.snoc (old r.contents) c }
  writes  { r.data.elts, r.put_ptr, r.contents }
=
  ...
```

Most of the specification is expressed in terms of the `contents` ghost field, which represents the sequence of elements currently stored in the buffer. The `snoc` function in the last postcondition is the concatenation of a single element at the end of a sequence. Indeed, if the result is 1, then the elements of the buffer are the same as before, except that the new element `c` was added at the end. Finally, the `writes` clause indicates the fields of `r` that may be written to by the function. Note that it specifies `r.data.elts`, rather than `r.data`. This means that while the function may write inside the zone pointed by `r.data`, the pointer itself (and crucially, the length of the pointed zone) is unchanged.

None of these postconditions on `contents` are difficult to prove, as `contents` is explicitly modified in the body of the function. Instead, the hard part is to prove that the type invariants of `r` are preserved, that is, that `contents` contains the elements that range from `r.get_ptr` to the new value of `r.put_ptr`. Two facts make this proof non-trivial. First, the fact that an element of `r.data` has been modified. We need to ensure not only that the element that is being added to `r.contents` is correct, which is easy enough, but also that we did not overwrite an element that was in the scope of `r.contents`. This would also be easy, if not for the second issue, which is the fact that all offsets are computed modulo `r.size`. This makes the problem much harder for automated solvers. The body of `ringbuf_put` is partially reproduced below. The proofs have been shortened for readability.

```
let ringbuf_put (ref r: ringbuf) (c: uint8) : int32
  ...
=
  if begin ensures { result = Seq.length r.contents }
    band_pow2 (sub_mod r.put_ptr r.get_ptr) r.mask r.exp
  end = (to_int32 r.mask) then return 0;
  C.set_ofs r.data (to_int32 r.put_ptr) c;
```

```
 8    mod_add1 (uint8'int r.put_ptr - uint8'int r.get_ptr) r.size;
 9    let ghost opp = pure { r.put_ptr } in
10    r.put_ptr ← of_int32 (band_pow2 (add' r.put_ptr 1) r.mask r.exp);
11    mod_submod (uint8'int r.put_ptr - uint8'int r.get_ptr)
12               (uint8'int opp + 1 - uint8'int r.get_ptr) r.size;
13    let newcontents = Seq.snoc r.contents c in
14    assert { mod (r.get_ptr + Seq.length newcontents - 1) r.size
15             = mod (r.get_ptr + mod (old r.put_ptr - r.get_ptr) r.size) r.size
16             = mod (old r.put_ptr) r.size
17             = old r.put_ptr };
18    assert { forall i. 0 ≤ i < Seq.length newcontents →
19             Seq.get newcontents i
20             = r.data[offset r.data + mod (r.get_ptr + i) r.size]
21             by if i = Seq.length newcontents - 1
22                ...
23                (* case split, 10+ lines *) };
24    r.contents ← newcontents;
25    return 1
```

Let us briefly discuss this rather involved proof. First, so as to not pollute the proof context, the computation of the number of elements in the buffer (lines 4-6) is done inside an abstract block (see Sec. 2.3.1). It has the same code and specification as the `ringbuf_elements` function, but the code is hidden from the rest of the proof, so the situation is the same as if we had simply called `ringbuf_elements`. However, the function call is inlined in the original C code, so we inline it as well. The `set_ofs` function, called at line 7, represents assignment through a pointer (with an offset). That line would be extracted to C as `r->data[(int32_t)r->put_ptr] = c`. The section between lines 8 and 12 computes the new value of `r.put_ptr` and proves that `r.put_ptr - r.get_ptr` has increased by one (modulo `r.size`). Finally, the largest part of the proof (lines 14-24) shows that `r.contents` is an accurate representation of the ring buffer elements. We split cases depending on whether we are considering the newest element of `r.contents` (the easy case), or one of the old ones, in which case we must prove that it was not modified. Again, most of the proof effort is spent proving simple facts on computations modulo `r.size`. This is not even the whole proof, as we had to prove four lemmas on modular arithmetic. The first two are instantiated automatically by the theorem provers during the proof of the last big assertion, and the last two are called manually (at lines 8 and 11). The `mod` function used in the specifications refers to the remainder of the Euclidean division.

```
let lemma mod_add (x y z:int)
  requires { z > 0 }
  ensures { mod (x + mod y z) z = mod (x + y) z }
= ...
  (* proof omitted for brevity *)

let lemma mod_add_compat (x y1 y2 z:int)
  requires { z > 0 }
  ensures  { mod y1 z = mod y2 z ↔ mod (x + y1) z = mod (x + y2) z }
= ()

let lemma mod_add1 (x z: int)
  requires { z > 0 }
  requires { mod x z < z - 1 }
  ensures  { 1 + mod x z = mod (x+1) z }
= mod_add x 1 z
```

```
let lemma mod_submod (x y z:int)
  requires { mod (x - y) z = 0 }
  requires { z > 0 }
  ensures  { mod x z = mod y z }
= mod_add (mod x z) (- mod y z) z
```

The proof of `ringbuf_get` is of the same ilk, although it is somewhat simpler as the function does not modify the elements of the buffer. Again, we spend a non-trivial amount of effort proving facts on modular arithmetic and the rest is very easy. For the sake of completeness, we reproduce the specification of `ringbuf_get` below. The function `Seq.([_..])` takes a sequence and returns its tail.

```
let ringbuf_get (ref r:ringbuf) : int32
  ensures { -1 ≤ result ≤ max_uint8 }
  ensures { result = -1 → r.contents = old r.contents
                       ∧ r.get_ptr = old r.get_ptr }
  ensures { result ≥ 0 → r.contents = Seq.([_..]) (old r.contents) 1
                       ∧ r.get_ptr = mod (1 + old r.get_ptr) r.size }
  ensures { Seq.length (old r.contents) > 0 ↔ result ≥ 0 }
  ensures { result ≥ 0 → int32'int result = Seq.get (old r.contents) 0 }
  writes  { r.get_ptr, r.contents }
```

We now have a complete, verified WhyML implementation of Contiki's ring buffer library. However, the astute reader may find that we have left important questions unanswered. Here is a non-exhaustive list.

- What exactly did we prove in terms of memory safety?
- Under what conditions can the type invariants be considered valid?
- What if the data buffer is also modified in some other part of the program?

Most of these questions stem from the fact that we still have not yet described our memory model. So, let us address the elephant in the room.

3.3 Memory model

There are various ways to model the C language's memory management, with various degrees of accuracy, expressiveness, and impact on our trusted computing base. At the beginning of this section, we take the perspective of the designer of such a memory model and review a few possibilities and the relevant tradeoffs. In Sec. 3.3.1, we consider a memory model based on a representation of the heap that uses explicit addresses. In Sec. 3.3.2, we sketch another memory model that relies on Why3's built-in alias tracking.

This second model is the one that we will use in the rest of the document. Sections 3.3.3 and 3.3.4 present improvements to the memory model that allow it to handle aliasing in a more fine-grained way. Finally, Sec. 3.3.5 explains how errors such as stack overflows or `malloc` failures are handled by the memory model.

3.3.1 An explicit memory model

A natural way to model the memory of a C program is to give all relevant objects an explicit representation that closely mirrors real-world implementations. For example, one could give explicit adresses to pointers, and model the heap and the stack as global arrays that map addresses to data. A very naive implementation of this idea might look like this:

```
type ptr ’a = abstract { addr: uint64 }

type mem ’a = abstract { mutable data: uint64 → ’a }

val ghost uint64_heap: mem uint64

val get (p: ptr ’a) (ghost heap: mem ’a) : ’a
  ensures  { result = heap.data[p.addr] }

val set (p: ptr ’a) (ghost heap: mem ’a) (v: ’a) : unit
  writes { heap.data }
  ensures { heap.data[p.addr] = v }
  ensures { forall a. a ≠ p.addr → heap.data[a] = old heap.data[a] }

val incr (p:ptr ’a) (ofs:int64) : ptr ’a
  ensures { result.addr = p.addr + ofs }
```

Of course, there are many issues with this naive model. First, the type `mem ’a` of heaps specifies the type of values that are stored inside. We would need to declare many global heaps, one for each value type. We would need to specify the correct heap whenever we call `get` or `set`, and make sure that the user cannot mistakenly create a value of type `mem` that is not one of our global heaps. Pointer type conversions would also be problematic. Another approach would be to use a single heap that represents a global array of bytes, and axiomatize the representation of each data type as bytes.

Also missing from this naive model is the notion of memory blocks. In the C standard, the heap memory is seen as a set of memory blocks called *objects*. The notion of object imposes restrictions on pointer arithmetic. For example, incrementing a pointer by an integer offset is only allowed as long as the result points to the same object (or one past its end). Therefore, the function `incr` is too permissive as it is. In order to be correctly specified, the function `incr` would actually need to be aware of the structure of the heap to properly enforce the restriction. Similarly, in order to properly specify dynamic memory allocation, `malloc` and `free` would need to be aware of which blocks have been allocated and/or freed.

Let us now assume that these various hurdles have been overcome and that we have a memory model that relies on an explicit representation of the heap. Several tools use memory models based on more refined versions of this approach, such as Frama-C/WP’s Hoare model [7]. Let us discuss the qualities and shortcomings of this approach.

One important thing to note with this explicit model is that pointers, which are essentially integers, contain no mutable fields. Therefore, they do not carry any region, so they cannot be affected by any of the alias tracking features of Why3 that are explained in Sec. 2.3.2. The notion of two pointers being aliased or not is meaningless. This is double-edged. On the one hand, bypassing WhyML’s aliasing restrictions allows many C programs to be implemented in WhyML. For instance, a C function may take two pointer arguments, test whether they are equal, and do different things depending on the result. This function is usually problematic because the arguments of a WhyML function are assumed to not be aliased, but it could be written using the explicit model.

In the end, the explicit model is quite expressive, because it closely mirrors an actual C implementation. On the other hand, using Why3's built-in alias control would decrease the specification and proof effort. Indeed, the explicit model does not provide very good framing properties. For example, the second postcondition of the function `set` is necessary for it to be usable in practice. One can imagine that when proving a non-trivial program, a meaningful amount of proof effort will be spent proving inequalities of addresses in order to prove that the contents of some objects did not change.

This is not even the only issue related to aliasing. Consider the ring buffer example again. Once a pointer has been passed to `ringbuf_init`, we would like to specify that it should not be used by anything else than the ring buffer. This cannot be easily done at the level of the specification of the ring buffer and its functions, because they are too local. We would need to extend the memory model to keep track of pointer ownership in some global way.

In the end, the approach consisting in modeling memory addresses explicitly is more difficult to put in practice than it may seem at first. To do so, one needs to explicitly model not only the memory layout and most of the memory management rules imposed by the C standard, but also some way to keep track of pointer ownership. Such a model would also involve a large number of axioms, which would need to be manually checked and then trusted, although most of these axioms might be close enough to the C standard to be easily understood and checked by an expert. The most important merit of this approach is that it can be used to write programs that could not be expressed under Why3's regular aliasing constraints. The largest issue is probably that it requires a way to solve the framing problem to preserve meaningful proof automation. There are many ways to solve this problem. A possibility would be try to encode some elements of separation logic into the model. Another would be to use a more refined representation such as F*'s hyper-heaps [91]. However, these methods would all require significant engineering, especially if we hope to generate goals that are well-handled by the SMT solvers. Considering that Why3's type system already has a built-in way to handle aliasing without encoding it into the goals that are passed to the external solvers, such an approach seems wasteful.

3.3.2 A block-based memory model

Let us now sketch another memory model that does rely on Why3's built-in alias tracking. It is a simplified version of the one I ended up adopting. Pointers should carry some mutable region. Two pointers are aliased if and only if the regions they carry are the same. The first thing to decide on is the degree of granularity. A naive adaptation from the explicit memory model might consist in storing in each pointer the contents of the memory cell it points to. This way, two pointers are aliased if and only if they point to the same cell.

```
type ptr 'a = { mutable data: 'a }
```

However, this is not really the right criterion to decide whether two pointers should be aliased, because of pointer arithmetic. Indeed, if the contents of some pointer `q` can be accessed through some other pointer `p` using pointer arithmetic, then they cannot be considered separated. Therefore, the simple definition above cannot work. Fortunately, there are restrictions on what one can do with pointer arithmetic in C. Indeed, starting from a pointer `p`, the only pointers that can be computed through pointer arithmetic are the ones that point inside the same memory block as p. Therefore, the right level of granularity for pointer aliasing is the memory block. This can be encoded in the Why3 type in a rather natural way.

```
type ptr 'a = abstract { data : array 'a ; offset : int }

val get (p:ptr 'a) : 'a
  requires { 0 ≤ p.offset < p.data.length }
  ensures { result = p[p.offset] }

val set (p:ptr 'a) (v:'a) : unit
  requires { 0 ≤ p.offset < p.data.length }
  ensures { p.data = (old p.data)[p.offset ← v] }
  writes { p.data.elts }
```

With this definition, each pointer stores the contents of the entire memory block it points at. The `data` field of a pointer is an array storing the block contents, and the `offset` field indicates which cell of the array it points at. Two pointers are aliased if their `data` arrays are the same, that is, if they have the same type including regions. In this case, performing an assignment through one pointer also updates the other. As expected, the primary way to obtain two aliased pointers is through pointer arithmetic.

```
val incr (p:ptr 'a) (ofs:int32) : ptr 'a
  requires { 0 ≤ p.offset + ofs ≤ p.data.length }
  ensures { result.offset = p.offset + ofs }
  ensures { result.data = p.data }
  alias { p.data with result.data }
```

The `incr` function returns the sum of a pointer and an integer offset. The precondition enforces that the result should point inside the memory zone pointed by `p` or one past its end. The `alias` clause specifies that the result is aliased with `p`, rather than simply carrying a copy of its data. The `alias` keyword is a somewhat recent addition to Why3. This work was the main reason it was added.

Note that the specification of `incr` allows the creation of pointers that point one past the end of the memory zone. This is what the C standard permits [53, 6.5.6 Additive Operators]. Such pointers cannot be dereferenced, but they may be used in comparisons. This is also why we do not use a type invariant in the definition of the `ptr` type to enforce that all pointers are valid. Invalid pointers (in the sense that they cannot be dereferenced) can still be computed in some situations, and it is sometimes useful to do so. The NULL pointer would be another special case. In the end, a type invariant would likely only eliminate the "`0 <= p.offset`" of `get` and `set`. It is more convenient to not have any.

Using the `incr` function, we can write functions for dereference and assignment with offset.

```
let get_ofs (p:ptr 'a) (ofs:int32) : 'a
  requires { 0 ≤ p.offset + ofs < p.data.length }
  ensures { result = p[p.offset + ofs] }
= get (incr p ofs)

let set_ofs (p:ptr 'a) (ofs:int32) (v:'a) : unit
  requires { 0 ≤ p.offset + ofs < p.data.length }
  ensures  { p.data = (old p.data)[p.offset + ofs ← v] }
  writes { p.data.elts }
= set (incr p ofs) v
```

Without belaboring the same points as the previous section too much, let us note again that many C programs cannot be expressed using this model due to the restrictions imposed by Why3's type system, but these restrictions make many specifications and proofs much simpler. In addition to the good framing properties provided by the aliasing restrictions, there is another somewhat unexpected upside to relying on Why3's alias tracking. A fairly large issue of the explicit model was the need to explicitly track which memory locations had been dynamically allocated (with `malloc`-like functions) and freed, as well as the boundaries of memory blocks. Such a tracking mechanism is needed to prevent issues such as double-free, use-after-free or out-of-bounds memory accesses. We did not explicitly sketch out what this tracking may look like, but we could imagine some global object mapping addresses to some type representing memory objects. The invariant would be that each memory block is associated to a distinct value of that type, with a special value representing unallocated blocks. It would then be relatively easy to check whether a pointer is valid, or whether two pointers are aliased. However, this represents quite a bit of work, especially because the specifications of functions using this model would also need to refer to this global state. One might notice that this description of a potential way to track aliasing and allocated blocks reads somewhat similar to Why3's region-based type system. Conversely, I would argue that Why3's type system natively provides most of what is needed to properly specify `malloc` and `free` without reinventing the wheel.

```
val malloc (sz:uint32) : ptr 'a
  ensures { result.data.length = 0 ∨ result.data.length = sz }
  ensures { result.offset = 0 }

val free (p:ptr 'a) : unit
  requires { p.offset = 0 }
  ensures  { p.data.length = 0 }
  writes   { p.data }

val is_not_null (p: ptr 'a) : bool
  ensures { result ↔ result.data.length > 0 }
```

The `malloc` function takes a positive integer as argument. It returns either a valid pointer of that length, or an invalid pointer (represented by a 0-length block). Implicitly, due to Why3's typing rules, the result is separate from all existing pointers, which is what we expect of `malloc`. Let us now explain the specification of `free`. As required by the C standard, it takes as argument a pointer to the beginning of a memory block. The postcondition renders its argument invalid, so it cannot be used anymore (the preconditions would not be provable). However, this is not sufficient to prevent any potential aliases from being used. This is performed by the `writes` clause, which indicates that the array at `p.data` has been replaced by some unspecified region. This induces a *reset* effect on it (Sec. 2.3.2). As an example, consider the following function.

```
let use_after_free (p:ptr int32) : int32
  requires { p.offset = 0 }
  requires { 2 ≤ p.data.length }
=
  let q = incr p 1 in
  free p;
  get q
```

It performs an illegal action. Indeed, it constructs a pointer `q` aliased to `p`, frees `p`, and attempts to use `q`. If we attempt to verify it, Why3 outputs the following error message, located on the expression `free p`: `This expression prohibits further usage of the variable q or any function that depends on it`. Note that this is a typing error, not a failure to prove some precondition in a WhyML program. This is exactly what we want. The specification of `free` ends up very simple, because the type system does all the heavy lifting.

As a side note, the same mechanism protects our ring buffer implementation from data races. Consider the following function. It calls `ringbuf_init` (passing it a pointer to a data buffer) and then attempts to write into the buffer. The same type error is raised. The reason is that `ringbuf_init` unifies the region of `r`'s data buffer with that of `p`. As the type of `p` was updated, it is subject to a reset. The only legal way to access that region afterwards is through `r`. The program would be legal if we replaced `set p 0` by a call to `ringbuf_put` (and accounted for the return type being `int32` instead of `unit`).

```
let data_race (ref r:ringbuf) : unit
=
  let p = malloc 32 in
  if is_not_null p then begin
    ringbuf_init r p 32 5;
    set p 0
  end
```

Finally, the specification of `realloc` combines those of `malloc` and `free`. If `realloc p sz` succeeds, `p` is invalidated, and a new pointer of length `sz` with the same contents as `p` (up to `sz`) is returned. If it fails, a null pointer is returned and `p` is left unchanged.

```
val realloc (p:ptr 'a) (sz:int32) : ptr 'a
  requires { 0 ≤ sz }
  requires { p.offset = 0 }
  requires { p.data.length > 0 }
  writes { p }
  writes { p.data }
  ensures { result.offset = 0 }
  ensures { result.data.length ≠ 0 → result.data.length = sz }
  ensures { result.data.length ≠ 0 →
            forall i:int. 0 ≤ i < old p.data.length ∧ i < sz →
              (result.data.elts)[i] = (old p.data.elts)[i] }
  ensures { result.data.length = 0 → p = old p }
```

It would be tempting to claim that this model enjoys a small trusted computing base, as the very short specifications of `malloc` and `free` seem to indicate. Unfortunately, this would be somewhat deceptive. The specifications are short, but the typing system imposes all sorts of implicit constraints. As a result, it is difficult to review the code manually and check the axioms. Again, the specification of `free` is a good example of this. On the other hand, one could argue that Why3 and its typing system would be part of the trusted computing base no matter what we do. However, this memory model tends to lean into the most intricate aspects of the typing system. Even for an experienced user, it can sometimes be hard to predict whether a program is well-typed or not.

The memory model I developed for this work was initially extremely similar to the block-based one that we just described. Over the course of the work of verifying a subset of GMP, it became apparent that this model was not quite expressive enough. While my current model is not so different from this one, I made a number of additions that allowed me to write a greater variety of C programs without running into type errors by manipulating pointer aliasing in a more fine-grained way.

3.3.3 Finer-grained aliases: splitting pointers

One of the main selling points of the memory model is that when working with two separate pointers, Why3 is aware that writing into one does not change the other, with no explicit user input required. However, some programs (such as quicksort implementations) split a single buffer into two separate areas and use the two halves as if they were separate. In this case, our memory model is not particularly helpful, and the user needs to perform a lot of extra work.

Worse, some programs that split buffers cannot be written at all using our memory model, because they attempt to use two halves of a pointer as parameters of the same function call. As all non-read-only regions passed to a function have to be separate, the program is rejected by Why3. An example of this can be found in Fig. 3.10. It is a sketch of a GMP algorithm presented in detail in Sec. 4.4.

```
1   let rec toom (r x y w: ptr uint64) (n: int32)
2   =
3     let h = n/2 in
4     let x’ = incr x h in
5     let y’ = incr y h in
6     let w’ = incr w n in
7     let r’ = incr r h in
8     let r’’ = incr r n in
9     ...
10    toom w r r’ w’ h; (* type error! *)
11    toom r’’ x’ y’ w’ h;
12    toom r x y w’ h;
13    ...
```

Figure 3.10: Ill-typed program that splits buffers.

The algorithm multiplies two large numbers `x` and `y` (represented as arrays of `n` machine integers, where `n` is assumed to be even in this simplified case). It takes two additional buffers of length 2`n`: `r` is meant to store the final result, and `w` is simply extra workspace. It is a divide-and-conquer algorithm, and calls itself three times on instances of size `n`/2. However, each half of each buffer serves various roles in the recursive calls. For example, the workspace `w` is split in two. In the recursive call at line 10, the first half stores an intermediary result, while the second half `w’` is used as workspace for subsequent recursive calls. However, the call at line 10 breaks WhyML’s aliasing rules. Indeed, `w` and `w’` are aliased, when function arguments should be separated. The program is ill-typed. However, while `w` and `w’` carry the same region, they are meant to cover separate slices of the same buffer, so in principle we would like them to be separate. I have modified my memory model to allow this, with three extra fields in the `ptr` type and a few more abstract functions. Figure 3.11 shows the new type and functions.

```
type zone

type ptr ’a = abstract {
  mutable data : array ’a ;
  offset : int ;
  mutable min : int ;
  mutable max : int ;
  zone : zone ;
}

(* Modified functions: min ≤ offset < max now required *)

val get (p:ptr ’a) : ’a
  requires { p.min ≤ p.offset < p.max }
  ensures { result = p[p.offset] }

val set (p:ptr ’a) (v:’a) : unit
  requires { p.min ≤ p.offset < p.max }
  ensures { p.data = (old p.data)[p.offset ← v] }
  writes { p.data }

(* New functions *)

val incr_split (p:ptr ’a) (i:int32) : ptr ’a  (* same extraction as incr *)
  requires { 0 ≤ i }
  requires { p.min ≤ p.offset + i ≤ p.max }
  writes   { p.max, p.data }
  ensures  { result.offset = p.offset + i }
  ensures  { p.max = p.offset + i }
  ensures  { result.min = p.offset + i }
  ensures  { result.max = old p.max }
  ensures  { result.zone = p.zone }
  ensures  { p.data.elts = old p.data.elts }
  ensures  { p.data.length = old p.data.length }
  ensures  { result.data.elts = p.data.elts }
  ensures  { result.data.length = old p.data.length }
  (* NOT alias result.data (old p).data *)

val join (p1 p2: ptr ’a) : unit               (* extracts to no-op *)
  requires { p1.zone = p2.zone }
  requires { p1.max = p2.min }
  writes   { p1.max }
  writes   { p1.data.elts }
  writes   { p2 }
  writes   { p2.data }
  ensures  { p1.max = old p2.max }
  ensures  { p1.data.length = old p1.data.length }
  ensures  { forall i. old p1.min ≤ i < old p1.max
                       → (pelts p1)[i] = old (pelts p1)[i] }
  ensures  { forall i. old p2.min ≤ i < old p2.max
                       → (pelts p1)[i] = old (pelts p2)[i] }
```

Figure 3.11: Memory model, with buffer splitting.

Let us explain these somewhat intimidating specifications. The pointer record now carries extra fields `min` and `max`. They represent ownership of a slice of the memory object. More specifically, they represent a promise that the program will not read or write outside these bounds through this pointer. The `data` field is guaranteed to be an accurate representation of the contents of the memory only within the [`min`, `max`) bounds. The specifications of `get` and `set` now require `min <= offset < max`. By default (such as when a pointer is allocated), `min` is 0, and `max` is equal to the block length, which corresponds to full ownership of the buffer. In order to call `free` on a pointer, it must also have full ownership.

Using these new fields, we can write a specification for a new incrementation function. The function `incr_split` has the same signature as `incr`. However, the result of `incr_split p n` is not aliased to `p`. They own two complementary slices of the buffer previously owned by `p`. That is, `p` is shrunk to only the sub-buffer left of `p+n`, while the result owns the slice that starts at `p+n`. It is analogous to Rust's `split_at_mut` function. The rest of the specification ensures that the contents of the two pointers are consistent at first. Since they are not aliased, if we write through one of the pointers afterwards, it will not update the contents of the other's `data` field. But this is fine, because the program cannot access the modified section of the data through the other pointer anyway. Therefore, the inconsistencies cannot be observed by the program. The `incr_split` function is extracted to C in the exact same way as `incr`.

Suppose we have called `let p' = incr_split p n in ...`, and done some writes on both `p` and `p'`. We have `min p <= max p = min p' <= max p'`. The array `p.data` accurately represents the memory contents between `min p` and `max p`, while `p'` accurately represents the other slice of the buffer. We get back to one pointer with full ownership of the buffer and accurate representation of its data using the function call `join p p'`. This invalidates `p'` and gives original ownership back to `p`. The last two postconditions of `join` ensure that `p.data` is accurate for the whole buffer. In the C code, there is no need for this operation (it only marks that the promise of keeping `p` and `p'` separate ends). Therefore, the `join` function is erased at extraction.

Obviously, `join` is meant to be used only on arguments that result from a call to `incr_split`, and would give wildly inconsistent results if it was used on two arbitrary pointers. This is enforced using the `zone` field of the pointer type. The `zone` type is fully abstract. Its semantics is that two pointers have the same `zone` if and only if they point to the same memory object. This is something of a meta-argument (and the first-order logic of Why3 does not allow us to express this), but the only way to obtain a proof that two pointers have the same zone is through the postcondition of `incr_split`.

Furthermore, after a call `join p p'`, `p'` and its aliases are invalidated. Indeed, the `writes` clause resets the region of `p'`, so the aliases of `p'` can no longer be used. As for `p'` itself, the type system allows it to be used, but the `writes` makes all its fields unknown, so the preconditions of the various pointer functions cannot be proved as far as `p'` is concerned.

These changes allow us to write a modified version of the function of Fig. 3.10 that is well-typed. As an added bonus, the fact that Why3 is aware that the sub-buffers are separate makes the proof simpler than if we had used a memory model that does not use Why3's built-in alias tracking.

```
let rec toom (r x y w: ptr uint64) (n: int32)
=
  let h = n/2 in
  let x' = incr_split x h in
  let y' = incr_split y h in
  let w' = incr_split w n in
  let r'' = incr_split r n in
  let r' = incr_split r h in  (* nested split:  r | r' |   r''   *)
  ...
  toom w r r' w' h;
  toom r'' x' y' w' h;
  toom r x y w' h;
  ...
  join r r';
  join r r'';
  join w w';
  join x x';
  join y y'
```

3.3.4 Finer-grained aliases: aliasing separate pointers

A common representation of the C heap is as a large, contiguous array. Our memory model, in which each pointer owns a separate section of the heap, can be seen as an abstraction layer over this representation. In some cases, this abstraction is not useful. One of these cases is writing functions whose parameters may or may not point to the same memory object.

Consider the following problem. We have a buffer `t` of `2n` bytes, and we want to swap the two halves of `t`, such that the contents of `t` are `t[n]`, . . . , `t[2n-1]`, `t[0]`, . . . , `t[n-1]`. We have an extra scratch buffer `w` of length `n` to store temporary values. In C, we could do this using the `memcpy` function. It takes a destination pointer, a source pointer, and a byte count, and copies that amount of bytes from the source to the destination. The pointers may not overlap, but they are allowed to point to independent sections of the same memory object. So, the following C code would solve the problem:

```
memcpy(w, t, n);
memcpy(t, t+n, n);
memcpy(t+n, w, n);
```

Can we write this in WhyML? The `memcpy` function can be implemented easily, with a simple loop. The issue is that in the second call to `memcpy`, the two pointers point inside the same memory object. This specific instance of the problem could be solved by splitting `t` using `incr_split`, but this is not sufficient in general (it would not help with cases where operands may overlap). We could also solve the problem in an ad-hoc way, such as by verifying two versions of memcpy (one for pointers that do not point into the same memory object, and one for those that do). For example, the version of `memcpy` that accepts only pointers to two distinct objects would have the following specification.

```
let memcpy_sep (dst src: ptr uint8) (sz: int32) : unit
  requires { sz > 0 }
  ensures  { forall i. 0 ≤ i < sz
                     → dst[dst.offset + i] = src[src.offset + i] }
  ensures  { forall j. j < offset dst ∨ offset dst + sz ≤ j
                     → dst[j] = old dst[j]}
  writes   { dst.data }
= ...
```

Using the alias keyword, we can write a specification that forces dst and src to have the same region (what is not allowed is writing a specification where they may or may not have the same region). Here is what such a specification could look like.

```
let memcpy_aliased (dst src: ptr uint8) (sz:int32) : unit
  requires { sz > 0 }
  requires { offset dst + sz ≤ offset src
             ∨ offset src + sz ≤ offset dst } (* no overlap *)
  alias    { dst.data with src.data }
  ensures  { forall i. 0 ≤ i < sz → dst[offset dst + i] = old src[offset src
    + i] }
  ensures  { forall j. j < offset dst ∨ offset dst + sz ≤ j
                       → dst[j] = old dst[j]}} (* also covers src *)
  writes   { dst.data } (* also covers src.data, which is the same region *)
= ...
```

This is enough to solve the problem (the first and third calls to memcpy should be calls to memcpy_sep, and the second to memcpy_aliased). However, this approach duplicates a lot of code. While the specifications differ in small ways, the code and proof of both memcpy versions should look extremely similar (the proof of memcpy_aliased should be a bit more involved, since the parameters are not separated by the type system). Moreover, the resulting extracted C code would contain two memcpy functions with very similar implementations, which would never happen in handwritten code. This is not a very far-fetched situation. In my verification of a fragment of the GMP library, this occurred quite frequently (see Sec. 4.2.2 for an example). So, I looked for a way to avoid duplicating work, as well as duplicating functions in the extracted code.

In a way, memcpy_aliased is the more general case. The alias keyword is needed to satisfy the type system, but of course the algorithm does not actually require dst and src to point inside the same memory object. Indeed, if dst and src are two separate pointers, nothing prevents us from imagining them encapsulated in a large buffer, with dst and src pointing to separate sections of that large buffer. This amounts to forgetting the abstraction performed by the memory model. This can be modeled using the abstract functions from Fig. 3.12. The basic idea is as follows. From two separate pointers x and y, we call open_sep and obtain two new pointers nx and ny, who are identical to x and y respectively, except that they point to non-overlapping sub-areas of some large memory object. We also obtain a ghost object m that carries enough book-keeping information to recover x and y using the close_sep function, as well as prevent unauthorized uses of close_sep.

Using these new primitives, memcpy_sep can be implemented without duplicating any code as a wrapper over memcpy_aliased, as follows:

```
let memcpy_sep (dst src: ptr uint8) (sz: int32) : unit
  requires { sz > 0 }
  ensures  { forall i. dst[offset dst + i] = src[offset src + i] }
  ensures  { forall j. j < offset dst ∨ offset dst + i ≤ j
                       → dst[j] = old dst[j]}
  writes   { dst.data }
=
  let nd, ns, m = open_sep dst src sz sz in
  memcpy_aliased nd ns sz;
  close_sep dst src sz sz nd ns m
```

At extraction, open_sep is replaced by a double identity function (as in, nd = dst and ns = src). Calls to close_sep are simply erased. We can ask the

```
type mem = abstract { zx: zone: zy: zone;
                      mix: int32; miy: int32;
                      max: int32; may: int32;
                      lx: int32; ly: int32; mutable ok: bool }

val open_sep (x y: ptr 'a) (sx sy: int32)
      : (nx: ptr 'a, ny: ptr 'a, ghost m: mem)
  requires { valid x sx ∧ valid y sy }
  requires { 0 ≤ sx ∧ 0 ≤ sy }
  ensures  { valid nx sx ∧ valid ny sy }
  (* nx and ny have the same data as x and y respectively *)
  ensures  { forall i. 0 ≤ i < sx →
             nx[offset nx + i] = old x[offset x + i] }
  ensures  { forall i. 0 ≤ i < sy →
             ny[offset ny + i] = old y[offset y + i] }
  (* nx and ny point to the same block *)
  ensures  { data nx = data ny }
  alias    { nx.data with ny.data }
  (* nx and ny do not overlap *)
  ensures  { offset nx + sx ≤ offset ny ∨ offset ny + sy ≤ offset nx }
  (* book-keeping information, for later recovery *)
  ensures  { m.zx = zone x ∧ m.zy = zone y }
  ensures  { m.mix = min x ∧ m.max = max x ∧ m.miy = min y ∧ m.may = max y }
  ensures  { m.lx = sx ∧ m.ly = sy }
  ensures  { x.data.length = old x.data.length
             ∧ y.data.length = old y.data.length }
  ensures  { pelts x = old pelts x ∧ pelts y = old pelts y }
  ensures  { m.ok }
  (* invalidate x and y and their aliases *)
  writes   { x, y }

val close_sep (x y: ptr 'a) (sx sy: int32) (nx ny: ptr 'a) (ghost m:mem) : unit
  requires { 0 ≤ sx ∧ 0 ≤ sy }
  (* nx and ny must be the output of a previous call to open_sep *)
  alias { nx.data with ny.data }
  requires { m.ok }
  requires { offset nx + sx ≤ offset ny ∨ offset ny + sy ≤ offset nx }
  requires { m.zx = zone x ∧ m.zy = zone y }
  requires { m.lx = sx ∧ m.ly = sy }
  (* recover the structure of x and y, with the new data from nx and ny *)
  ensures { m.mix = min x ∧ m.max = max x ∧ m.miy = min y ∧ m.may = max y }
  ensures { forall i. 0 ≤ i < sx →
             x[offset x + i] = old nx[offset nx + i] }
  ensures { forall i. 0 ≤ i < sy →
             y[offset y + i] = old ny[offset ny + i] }
  ensures { forall j. j < offset x ∨ offset x + sx ≤ j
                → x[j] = (old x)[j] }
  ensures { forall j. j < offset y ∨ offset y + sy ≤ j
                → y[j] = (old y)[j] }
  ensures { x.data.length = old x.data.length
            ∧ y.data.length = old y.data.length }
  writes  { x, y}
  (* invalidate m, as well as nx, ny and their aliases *)
  writes { nx, ny, m.ok }
```

Figure 3.12: Aliasing separated pointers.

extraction mechanism to inline calls to memcpy_sep, so there is only one memcpy function in the extracted code. So, we have managed to avoid duplicating proof work, and made the extracted code more idiomatic. But at what cost? The open_sep and close_sep specifications are somewhat reusable (I still needed to write separate versions with arity 3), but they are very long and error-prone. Giving even a handwavy justification of their consistency is daunting. Some elements, such as invalidating the unsafe aliases of the various pointers involved, are not explicit in the specification. When reviewing this type of specification, one needs to remember that seemingly innocuous writes clauses can completely change the meaning of the function. In the end, while this ad-hoc solution was somewhat workable, it served to show that this memory model really is best suited to verifying programs where aliasing is not a big concern. In order to verify aliasing-heavy programs, it would be better to switch to a model such as the explicit one from Sec. 3.3.1.

3.3.5 Error handling

C programs can fail in a number of ways. Some of these failures, such as out-of-bound array accesses, can be attributed to bugs in the program. We should expect proved WhyML programs to not contain them. However, even well-behaved, bug-free C programs can fail more or less gracefully. For example, malloc can run out of memory and return a null pointer, in which case the program should detect this, and then deal with the failure (usually by reporting the error and aborting). Other low-level functions may fail in even less recoverable ways. For example, the alloca function allocates memory on the stack. Its man page states: “If the allocation causes stack overflow, program behavior is undefined.” In practice, it is likely that a segmentation fault occurs and the program crashes. So, how to deal with this in the memory model?

There are multiple ways to handle errors in traditional WhyML programs. The most important one is to prevent them from happening using function preconditions. For example, rather than specifying that division can fail when the divisor is 0, we can simply require the divisor to not be 0 in preconditions. Other common errors such as out-of-bounds array accesses tend to be forbidden by preconditions. Why3 also supports OCaml-style exceptions. For example, Why3’s standard library contains two functions to get an array element. They are reproduced below.

```
val ([]) (a: array 'a) (i: int) : 'a
  requires { [@expl:index in array bounds] 0 ≤ i < length a }
  ensures  { result = a[i] }

let defensive_get (a: array 'a) (i: int)
  ensures { 0 ≤ i < length a ∧ result = a[i] }
  raises  { OutOfBounds → i < 0 ∨ i ≥ length a }
= if i < 0 || i ≥ length a then raise OutOfBounds;
  a[i]
```

The specification for ([]) is straightforward. On the other hand, it may seem surprising that defensive_get ensures that its argument is within the bounds of the array. The explanation is that the postcondition only covers the case where the function returns normally (as opposed to raising an uncaught exception). In this case, i is indeed in the array bounds (otherwise the exception would have been raised). The raises clause handles the case of the function exiting through an uncaught exception. It may or may not be caught by the caller, just like in an OCaml program. However, any function that calls `defensive_get` must either catch the exception, prove that it is not raised, or have its own `raises` clause that specifies the fact that the exception might escape.

None of these approaches is particularly well-suited to modeling a program dealing with a `malloc` failure. On the one hand, there is no good precondition to give to `malloc` to ensure it does not fail. On the other hand, the concept of exceptions does not map to C very well. Certainly, one could imagine declaring a special exception named `Abort`, and replacing `raise Abort` with a call to the `abort` function in the extracted C code. However, this exception would never be caught, and there would be no way to prove that it is not raised. So, the specification of any C program that uses `malloc` (or that calls functions that use `malloc`, and so on) would need to be polluted by an extra clause such as `raises { Abort -> true }`. So, the exception-based approach is unsatisfying.

Another possible approach is to use non-termination as a model of a program abort. WhyML functions can be annotated with the `diverges` keyword, which signifies that the function may not terminate. It is necessary to use it for functions that contain an infinite loop, for example. Functions annotated with this keyword cannot be used in C code. Their callers must also explicitly be marked as non-terminating. This can be used to succintly model C's `assert` function:

```
val c_assert (e:bool) : unit
  ensures { e }
  diverges
```

The behavior of C's `assert` function is as follows: if the condition is true, nothing happens, otherwise the program aborts. The `c_assert` function only terminates if its argument is true, so replacing it by `assert` is appropriate in the sense that the extracted C program can have no behavior that does not occur in the WhyML program. We can use this to write a WhyML program that calls malloc and checks its result. The resulting extracted code would be a function that calls `malloc` and then `assert` on the result, which is a valid way to check the result of `malloc`, if not the most idiomatic.

```
let malloc_checked (sz:uint32) : ptr 'a
  ensures { result.data.length = sz }
  ensures { result.offset = 0 }
  diverges
=
  let p = malloc sz in
  c_assert (is_not_null p);
  p
```

The only issue is that, again, any callers of `c_assert` would need to be marked as diverging. This is not ideal, and not only because we do not want to add an extra keyword to specifications. If a function is marked as diverging, Why3 no longer requires to prove that, for example, its loops all terminate. We ended up adding a new keyword to WhyML to resolve this. Functions can now be marked as `partial` to specify that they may fail due to external factors, or more generally, that they have observable effects that are not represented in their specification. (Arguably, crashing is a side effect.) The `partial` keyword represents an intermediary state between regular program functions (total termination, can be used in ghost code) and diverging functions (cannot be used in ghost code, all callers must also be marked as diverging). Partial functions cannot be used in ghost code. Their callers are also partial, but do not need to be explicitly marked as such in their specifications. Finally, loops inside partial functions still require a proof of termination. In fact, the source of partial termination is necessarily a `val` function (that is, without a body). The only way to obtain a partial function with a body is to make it call another partial function.

The final specification of `c_assert` is as follows:

```
val partial c_assert (e:bool) : unit
  ensures { e }
```

Similarly, our model of `alloca` is partial and its specification assumes that it always succeeds. This is a valid model if we make the assumption that a stack overflow immediately crashes the program, rather than causing more subtle failures.

3.4 Extraction

This section explains the process of compiling WhyML programs to idiomatic C code. We start by explaining the basics of Why3's extraction mechanism (Sec. 3.4.1) and laying out the design choices that led the development of the C extraction (Sec. 3.4.2). Section 3.4.3 explains the extraction of basic WhyML constructs such as function calls and the let-in construct. Section 3.4.4 provides some details on extraction drivers. In Sec. 3.4.5 through 3.4.8, we explain how various WhyML constructs are extracted: control flow structures, tuples, mutable records, and arrays. Section 3.4.9 describes how libraries comprised of multiple files are extracted. In Sec. 3.4.10, we describe a number of features aimed at increasing the readability of the extracted code. Finally, in Sec. 3.4.11, we evaluate the extraction on our running example, Contiki's ring buffer library.

3.4.1 Why3 extraction basics

Let us first briefly describe Why3's extraction mechanism. It involves an intermediate language called ML. ML has a similar syntax to WhyML, but has a much simpler type system (side effects and regions are not a part of it), no specifications and assertions, and no ghost code. Why3's extraction is a two-step process. First, the WhyML program is compiled to ML. In this step, all ghost code and logical content (specifications, assertions, lemmas, and so on) are erased. Ghost fields are eliminated from records, ghost arguments are eliminated from functions, and so on. A few minor transformations are performed, such as optimizing away records with a single non-ghost field. The `val` declarations (that is, functions without a body) are also erased. The translation is otherwise straightforward for non-ghost code. This step is the same for all target languages.

The first step outputs an ML abstract syntax tree. The second step consists in printing source code in the target language. For each available target language, a corresponding printer is registered. A printer consists in a set of functions. The main function in a printer takes a ML declaration (which can be a type declaration or a function definition) and prints it in the target language. The other functions are auxiliary ones that print file headers, footers and so on.

This two-step architecture may read overly simplistic, and the name "printer" sound like an odd choice. Indeed, the second step covers both the translation from ML into the target language and the source code production. The reason for this choice is that at the beginning of this work, only OCaml was available as a target language (CakeML was added later on). ML is very close to being a sublanguage of both OCaml and CakeML, so in both of these cases, the translation from ML to the target language is trivial and the name "printer" is appropriate. One of the contributions of this work is the addition of C as a target language for Why3's extraction. As you can imagine, the C "printer" is much more involved than the other two.

The translation from WhyML to ML is documented in Mário Pereira's thesis [79] and is not specific to C extraction. The rest of this section focuses on describing the process of compiling an ML abstract syntax tree to C code. This process was outlined in a previous article [85]. Although many new features have since been added, the underlying principles have not changed.

3.4.2 Design choices, supported WhyML fragment

The extraction from WhyML to C was designed in the context of verifying efficient C programs such as the GMP library. This informs the design choices in a few ways. First, it is important that the extraction process does not introduce inefficiencies in the extracted code that were not present in the WhyML source code, for example by adding extra indirections and closures or by using a garbage collection library. Ideally, the WhyML programmer should be able to predict what the extracted code will look like. On the other hand, it is acceptable to not be able to extract every existing WhyML program to C. What matters is being able to implement most C programs in WhyML and extract them back to C. Finally, the extraction process itself is not verified. Therefore, it must be as simple as possible, the goal being to decrease the impact on the trusted computing base as much as possible. The extraction does not need to perform many optimizations. Indeed, the extracted C code is meant to be compiled in turn by an optimizing compiler, which we are very unlikely to outperform. Any optimizations performed by the extraction should be there to make the extracted code more readable, not more efficient. These soft constraints all point towards the same direction, which is to make the translation from ML to C as straightforward as possible, while rejecting most WhyML constructs that do not lend themselves well to being straightforwardly translated to C.

Here is a non-exhaustive list of WhyML features that are rejected by the C extraction:

- higher-order functions and types,
- polymorphism (except in driver functions, see Sec. 3.4.4),
- exceptions, `try...with` constructs,
- unbounded mathematical integers and real numbers.

Sum types (such as the `list` type from Sec. 2.1.3) are also currently unsupported, although the non-recursive ones could be extracted as C unions in principle. The WhyML source code is expected to handle heap allocations and deallocations explicitly using a memory model such as the one from Sec. 3.3.

Therefore, the extraction itself does not need to care about memory management. There is one exception, which is checking that adresses on the stack do not escape their scope. This is discussed in Sec. 3.4.7.

3.4.3 Basic constructs

With the problematic WhyML constructs out of the way, we expect the translation from WhyML to C to be relatively straightforward. This section aims to explain what this means on a few basic constructs. Let us first introduce a few notations. While they may look like formal notations, the goal is not to provide a formal proof of the extraction mechanism, but merely to informally explain how it works in a more concise way.

The main function in the C extraction takes a WhyML expression e and an environment Γ, and returns a pair `(d,s)` where `d` is a list of typed C variable definitions and `s` is a C statement. Let us denote this function using the double bracket notation $[\![.]\!]_\Gamma$. The semantics of the pair `(d,s)` are as follows: if $[\![e]\!]_\Gamma = (((v_1, t_1), \ldots, (v_n, t_n)), s)$, then e is extracted to the following C block:

```
{
  t1 v1;
  ...
  tn vn;
  s;
}
```

Note that s may be a sequence of semi-colon-separated statements. By a similar abuse of notation, we abridge this block to the following, although `d` may include many variables of different types.

```
{
  d;
  s;
}
```

The environment Γ contains information such as whether the current expression is in terminal position, the name of the current function, and other similar information that will be made more explicit later on. It will be omitted from the notations when not relevant. Let us now outline a few basic constructs.

Let-in construct If the expression e_1 extracts to `(d1, s1)` and e_2 extracts to `(d2, s2)`, how to extract `let (x:t) = e1 in e2`? In this case, we need to perform a minor program transformation. By construction, `s1` is a statement that computes some value v that has the same type `t` as x. We transform this statement into one that assigns v to the variable x, which is assumed to have been declared earlier on. The general idea of the transformation is to push the assignment to the leaves of the syntax tree that are in tail position. Let us call this transformation $A(x, .)$ and give a few examples:

- $A(x,$`e`$)$, where `e` is an expression, is `x=e`.
- $A(x,$`if c then e1 else e2`$)$ is `if c then` $A(x,$`e1`$)$ `else` $A(x,$`e2`$)$
- $A(x,$`{s1; s2}`$)$ is `{s1;` $A(x,$`s2`$)$`}`
- $A(x,$`let v = e1 in e2`$)$ is `let v = e1 in` $A(x,$`e2`$)$

With $A(x, .)$ defined, we can extract the let-in construct to the following:

```
{
  t x;
  {
    d1;
    A(x, s1);
  }
  {
    d2;
    s2;
  }
}
```

The end result is quite a bit longer than the original code, notably because the code is split between several sub-blocks. However, the only reason to do so is the potential presence of name conflicts between x and the variables in `d1` and `d2`. Thankfully, Why3 prevents such conflicts. Indeed, WhyML variables all have unique names internally. When variable shadowing should occur, instead one of the variable gets a fresh name. For example, the WhyML program `let x = 4 in let x = 5 in x` could be represented as `let x1 = 4 in let x = 5 in x` internally. The end result is that all the variables in the definition are distinct, so the extracted code can be shortened to the following, much more readable block. More generally, throughout the extraction process, some simple program transformations take care of lifting all variable definitions to the top of the function and flattening the block structure.

```
{
  t x;
  d1;
  d2;
  A(x, s1);
  s2;
}
```

Finally, note that I did not mention how the WhyML type of x is translated to the C type t. Most WhyML types that are meant to be extracted are simply listed in the extraction driver explicitly (see Sec. 3.4.4). Indeed, one could imagine that some basic types could be hard-coded into the extraction function, but there are very few built-in Why3 types (most are defined in user theories). Many WhyML built-in types, such as the type `int` of unbounded mathematical integers, cannot be extracted to C at all. In the end, the function that converts WhyML types to C types mostly handles the composite types, such as tuples and records (Sec. 3.4.7), arrays and references (Sec. 3.4.8). The simple types are left up to the driver, with the exception of boolean constants.

Toplevel expressions, and the C `return` statement In WhyML, the body of a function is an expression that computes a value. The body of a C function generally cannot be just an expression. The value that is computed should be returned using the `return` statement. In terms of extraction, this is somewhat similar to the let-in case, in that we need to transform an expression that computes a value into a statement that returns it. This could be expressed using the transformation A from last section, by first transforming the expression `e` into something like "`let result = e in return result`". However, we do not need to do anything so complex, a simple boolean in the environment Γ

is enough. Essentially, when an expression is in tail position, we should add a `return` statement, and if it's not (such as the inner expression of a let-in construct, or the guard of a loop), then we should not.

Let us take the simplest expression as an example. If v is a variable name, then $[\![v]\!]_\Gamma = (\emptyset, \texttt{return v})$ if v is in terminal position and $(\emptyset, \texttt{v})$ if not. When extracting a more complex expression, such as `let v = e1 in e2`, then for the extraction of the subexpression `e1`, we update Γ to mark that we are not in tail position. The subexpression `e2` is in tail position if and only if the whole expression is.

Booleans, `if` statements WhyML features a boolean type with two elements `True` and `False`. The C standard defines a `_Bool` type, which is the smallest integer type and contains 0 and 1. However, C `if` statements and loops take scalars (which can belong to the `_Bool` type, but also be other integers, pointers, or even floating-point numbers) as their test expressions. If the expression evaluates to 0, then the condition is false, otherwise it is true. We do not have a good reason to force the `bool` type to be translated to any particular C scalar type, so this is left up to the driver.

However, being able to extract WhyML code that contains the literals `true` and `false` is useful too, and their translations cannot be specified in the driver for technical reasons (they are WhyML keywords rather than library functions). In the end, we extract `true` to the literal 1 and `false` to the literal 0. This effectively forces the translation of the `bool` type to be either `_Bool` or some other integer type, rather than a floating-point or a pointer type. In the default driver, we choose `int` because it seems to be the most widely used in practice for historical reasons. In principle, it is possible to use another integer type.

Note that giving an explicit translation to `bool` is only needed if it explicitly appears in the source code, that is, if the user wants to extract a WhyML function that returns a boolean or takes one as parameter. The mere presence of `if` constructs in the code does not require giving a translation to `bool`.

Let us now discuss how to extract the expression `if e1 then e2 else e3`. If $[\![\texttt{e1}]\!]_\Gamma$ is simple (e.g., an expression), the whole expression can simply be extracted transparently to a C `if` statement. If it is more complex (e.g., another `if` statement), then we compute it beforehand. More explicitly, if $[\![\texttt{e1}]\!]_{\Gamma'} = \texttt{(d1, s1)}$ (the difference between Γ and Γ' being that `e1` is not in terminal position in Γ'), $[\![\texttt{e2}]\!]_\Gamma = \texttt{(d2, s2)}$ and $[\![\texttt{e3}]\!]_\Gamma = \texttt{(d3, s3)}$, then `if e1 then e2 else e3` is extracted as follows.

```
{
  d1;
  int cond;
  A(cond,s1);
  if (cond) {
    d2;
    s2;
  }
  else {
    d3;
    s3;
  }
}
```

We declare `cond` as a fresh WhyML identifier beforehand to avoid any name conflict. Note that `cond` has type `int`. This imposes additional constraints on the C translation of the `bool` type, if the user cares to define one. Indeed, it needs to have a conversion rank smaller than that of `int`, so that it can be converted to `int` implicitly. In practice, this means that the user may define the translation of `bool` as any integer type that is not strictly wider than `int`.

Function call (default case) Consider the function call `f(e1)`, where `e1` is an arbitrary ML expression. Assume `f` is a regular function defined elsewhere in the WhyML program, and that it has already been extracted to a C function also named `f`. Simply translating this function call to $\texttt{f}(\llbracket\texttt{e1}\rrbracket)$ is not correct in general, as $\llbracket\texttt{e1}\rrbracket$ may not be a simple C expression. In the general case, we compute the function arguments before the call to `f`. Suppose `e1` has type `t` and $\llbracket\texttt{e1}\rrbracket = (\texttt{d}, \texttt{s})$. Then the call may be extracted as follows.

```
{
  d;
  t x1;
  A(x1, s);
  f(x1);
}
```

If there are several function arguments `x1`, `x2` and so on, they are computed sequentially before the function call.

Several transformations can be applied to make the extracted code more readable. For example, if `d` is empty and `s` is in fact a simple C expression, then the extracted code is simply `f(s)`. More complex transformations are discussed in Sec. 3.4.10.

3.4.4 Extraction drivers

Extraction drivers are text files which are passed to the extraction command alongside the source WhyML files. They allow users to customize the output of the extraction to a large degree. Their most common use case is to provide code for WhyML `val` declarations. Drivers are not an original contribution of this thesis. However, the precedence system presented later in this section is.

Function calls

Consider the following WhyML program on 64-bit unsigned integers.

```
type uint64 = < range 0 0xffffffffffffffff >

val add (x y:uint64) : uint64
  ensures { uint64'int result
            = mod (uint64'int x + uint64'int y) (uint64'maxInt + 1) }

let double_and_add (x:uint64) = add x (add x 42)
```

The program is based on our model of 64-bit unsigned integers from Sec. 3.1. The `uint64` type is declared as a range type and the integer addition is modeled with the uninterpreted function `add`. In the extracted code, the goal is to seamlessly translate `uint64` into the C type `uint64_t`, and to replace calls to `add` by the C operator `(+)`. This is specified in the driver file as follows.

```
syntax type uint64 "uint64_t"
syntax val add     "%1 + %2"
```

The semantics of the `syntax type` directive are transparent: it states that the WhyML type `uint64` should map to the C type `uint64_t`. The second directive states that calls to `add` should be replaced by the quoted expression, where `%1` is the first argument and `%2` is the second one. Just like in the previous rule for function calls, if the arguments are not simple expressions, they are computed apart from the function call.

In the end, the extracted code for the example program is as follows.

```
uint64_t double_and_add(uint64_t x) {
  return x + (x + 42)
}
```

Precedences

Note the parentheses in the above example. The `return` statement could just as well be written "`return x + x + 42`". The reason for these parentheses is simple. The contents of the driver strings is opaque to the extraction mechanism (the argument substitution is performed at the last moment, when we are printing the extracted code). Therefore, the printer is unaware of the precedences of the expression being printed, and has to add parentheses everywhere. In this example, it is not that bad, because the parentheses around variables and constants are automatically removed, but in more complex programs it sometimes makes the extracted code quite ugly and hard to read.

In order to improve the readability of extracted code, I have added to Why3 a mechanism that allows drivers to specify the precedences of an expression and its subexpressions. For example, the directive for `add` may be as follows:

```
syntax val add "%1 + %2" prec 4 4 3
```

The first number is the precedence level of the expression. Lower numbers indicate higher precedence. The remaining numbers indicate the precedence level that is required for each subexpression to be printed without parentheses. In this case, the whole expression is at precedence level 4, while the two subexpressions `%1` and `%2` require parentheses if they are at precedence level above 4 and 3 respectively (the `(+)` operator is left-associative, so the right subexpression needs higher priority). The precedence levels of many operators (such as `=`, `==` and so on) are hard-coded in the extraction mechanism.

At first glance, it may appear that this system for specifying precedences is needlessly complicated. For binary operators, it would be enough to simply specify a precedence level for the whole expression, as well as associativity information (left, right, or none). This is what Coq does in the `Notation` command. For example, a Coq user may introduce a binary operator with the following syntax: `Notation "A /\ B" := (and A B) (at level 80, right associativity).`

The reason why we allow a more complex specification for precedences is that we want to allow parentheses to be printed even when they are not strictly necessary, in the event that it makes the extracted code more readable. Examples of this can be found in Sec. 3.4.10.

Preludes

Drivers also feature a `prelude` directive. It is followed by a string, which is printed at the top of the extracted code. There are two common use cases.

The first is system includes. For example, the driver may include a line such as `prelude "#include <stdint.h>"`, in order to use the types `int32_t` and so on. The second use case is defining functions and variables that are referenced in driver directives. In the example below, `lookup` is a `val` declaration in some WhyML program that represents a precomputed lookup table. The lookup table is hardcoded in the driver prelude (the driver prelude could also be a `#include` directive to a handwritten C file where the table is defined).

```
prelude "int __lookup_table[255] = { ... }"

syntax val lookup "__lookup_table[%1]"
```

Polymorphism

Driver directives allow a large amount of expressivity. For example, they enable polymorphism to a small degree. This is how we can give a polymorphic type to pointers and the `malloc` function. Recall the signature of our WhyML model of `malloc`:

```
type ptr 'a = ...

val malloc (sz: uint32) : ptr 'a
```

We do not want to extract a polymorphic type declaration to C. Moreover, a call to `malloc` in C should take into account the size of the elements of the array to be allocated. For example, suppose we want to allocate an array `p` of five elements of type `uint8`. In WhyML, we could write:

```
let p : ptr uint8 = malloc 5 in
...
```

In C, a correct translation would be:

```
p = malloc(5 * sizeof(uint8_t))
```

The driver mechanism is flexible enough to allow this without needing to declare a separate `malloc` function for each type. The following is an excerpt from the default driver for C extraction.

```
syntax type ptr "%1 *"
syntax val malloc "malloc(%1 * sizeof(%v0))" prec 1 3
```

The `syntax type` directive is similar to the usual function directives. Each instance of `ptr 'a` is extracted to the correct type by substitution of the type variable. Note that the polymorphic type `ptr 'a` still cannot be extracted if the type variable is not instantiated. The `syntax val` directive for `malloc` makes use of another feature of the driver syntax. In addition to referring to the function arguments using `%1` and so on, one may refer to their types using `%v1` and so on. The `%v0` identifier refers to the return type.

Drivers are a powerful tool that allows users to customize program extraction with a lot of expressivity. However, keep in mind that the contents of drivers are not verified, and that they are implicitly trusted by the extraction. Ideally, drivers are short enough to be reviewed by hand, much like the axioms in the WhyML formalizations. In my experience, however, a majority of bugs in my extracted code came from errors in my driver.

3.4.5 Control structures

This section goes over a few classical control structures and explains how to translate them from ML to C.

While loop

ML features a classical `while e1 do e2 done` construct, with `e1` and `e2` being two expressions. If the loop condition ⟦`e1`⟧ is a simple expression, the whole loop can be translated straightforwardly to `while (`⟦`e1`⟧`) {` ⟦`e2`⟧ `}`. If ⟦`e1`⟧ is more complex, this does not work. In that case, suppose ⟦`e1`⟧ = `(d,s)`. We give the loop a trivial condition, compute ⟦`e1`⟧ at the start of each iteration, and immediately break out of the loop if it is false.

```
d;
int c;
while (1) {
  A(c,s);
  if (!c) break;
  ⟦e2⟧
}
```

Break and return

While C does not have exceptions in the same sense as OCaml, several C constructs serve a similar purpose in altering a program's control flow. The `break` statement can be used to exit a loop early, and the `return` statement can be used to exit a function call early. When one wants to translate a C program that uses these features into a WhyML program, the only choice is to emulate the behavior of `break` and `return` with exceptions. The following sketches of WhyML programs emulate `break` and `return` with the exceptions `B` and `R` respectively.

```
exception B                        exception R of t

try                                let f (...) : t =
  while ... do                     ...
    ...                            try
    if (...) then raise B;           ...
    ...                              raise (R e)
  done                               ...
with B → ()                        with R v → v
end                                end
```

The extraction mechanism recognizes both of these patterns and replaces them by uses of `break` and `return` respectively. More precisely,

$$\llbracket \texttt{try while e1 do e2 done with B -> () end} \rrbracket_{\Gamma} = \llbracket \texttt{while e1 do e2} \rrbracket_{\Gamma'}$$

The environment Γ' carries the information that `raise B` should be extracted as `break`. Note that the name `B` of the exception is not hard-coded. Any exception of arity 0 works. The pattern for `return` is detected in the same way, with the additional condition that the current expression is in tail position (rather than the expression inside a let-in, for instance). All other instances of `try` or `raise` are unsupported by the C extraction and rejected. Note that WhyML has recently added `break` and `return` keywords in the surface language, but they are syntactic sugar for these patterns.

For loop

WhyML also features `for` loops such as `for v = l to u do e done`, where the variable `v` is a bounded integer. One can also iterate downwards by using `downto` instead of `to`. Note that `v`, `l`, and `u` must be simple variables, not arbitrary expressions (the user can write arbitrary expressions for `l` and `u`, but they are normalized by Why3 before extraction). This means that we do not have the usual issue where `l` and `u` may be too complex to inline. So, can we translate this to `for (v = l; v <= u; v++) {` ⟦e⟧ `}`? Sadly, we cannot. This time, the issue is the boundary conditions. For example, if `u` is the largest integer of its type, the condition `v <= u` is always true, and `v++` will overflow. We can replace it by a test inside the loop that breaks out if `v = u`, but then we also have to add a test at the beginning of the loop, to prevent the first iteration from being performed if `l > u`. In the most general case, if `v` has type `t`, the translation is as follows.

```
t v;
if (l <= u) {
  for (v = l; ; v++) {
    ⟦e⟧
    if (v == u)
      break;
  }
}
```

If extra information about the bounds is available (for example, they may be literals), some optimizations remove the tests when able so as to make the code more readable.

3.4.6 Tuples

WhyML programs regularly involve functions that return multiple values. In these cases, the function typically returns a tuple. C does not have the equivalent of WhyML's polymorphic tuple. A natural approach is to translate tuples into C structs. Tuples are essentially a special case of immutable records.

For each function `f` that returns a tuple, we declare at the toplevel a struct named `__f_result`. It has a field for each element in the function result. When extracting a tuple `(x, y)`, if we are in tail position, the translation goes as follows. First, we declare a struct that has the type of the result of the current function (carried in the environment Γ). We populate its fields with the tuple elements, and then we `return` the struct. A complete example can be found in Fig. 3.13. The result is much longer than the WhyML code, but it is efficient, as an optimizing compiler will typically store the struct fields on registers.

3.4.7 Mutable structures

There are a number of way to use mutable data structures in WhyML programs. The three main ones are mutable record fields, arrays, and references. Let us focus on references for now. They are declared with the `let ref` keyword, and they are essentially records with a single mutable field. In C, all variables are mutable, so a natural thing to do could be to extract WhyML references to regular C variables, just as if they were not references. For example, the

```
let f (x:int32) : (int32, int32)
  = x + 1, x + 2

let g (x:int32) =
  let (y, z) = f x in
  y + z
```

```
struct __f_result {
  int32_t __field_0;
  int32_t __field_1;
};

struct __f_result f(int32_t x) {
  struct __f_result result;
  result.__field_0 = x + 1;
  result.__field_1 = x + 2;
  return result;
}

int32_t g(int32_t x) {
  int32_t y, z;
  struct __f_result struct_res;
  struct_res = f(x);
  y = struct_res.__field_0;
  z = struct_res.__field_1;
  return y + z;
}
```

Figure 3.13: Extraction of a WhyML program involving tuples.

```
let set (ref x: int32) : unit =
  x ← 42;
  ()

let main ()
  ensures { result = 42 }
=
  let ref x = 0 in
  set x;
  let ref y = 0 in
  y ← 42;
  x + y
```

```
void set(int32_t * x) {
  *x = 42;
}

int32_t main() {
  int32_t x, y;
  x = 0;
  set(&x);
  y = 0;
  y = 42;
  return x + y;
}
```

Figure 3.14: Unboxing WhyML references.

expression `let ref x : int32 = 0 in x <- 1` could be extracted as the C block `int32_t x = 0; x = 1`.

Function calls require special attention. In WhyML, all mutable structures are implicitly passed by pointer. The extraction needs to box mutable structures that are passed as function arguments, keep track of them in the environment, and dereference them as needed. Figure 3.14 shows a complete example of extraction of a program involving mutable variables. The variable `x` is passed to `set` by reference.

Another possible issue is that stack addresses might escape their scope. For example, a function might declare a reference and return it. In the extracted C code, the function would return a memory address pointing to an invalidated section of the stack. Rejecting this specific pattern is not enough, as stack addresses could still escape in more roundabout ways (as a struct field, for example). Performing a complex escape analysis is a bit daunting for an extraction mechanism that was supposed to be as simple as possible. Thankfully, Why3's

```
type t = { x : int32; y : int32 }

let f (a:t) =
  { x = a.x + a.y; y = a.x - a.y }
```

```
struct t {
  int32_t x;
  int32_t y;
};

struct t f(struct t * a) {
  struct t t;
  t.x = a->x + a->y;
  t.y = a->x - a->y;
  return t;
}
```

Figure 3.15: Extraction of WhyML records.

type system comes to the rescue. Indeed, the type of an expression `e` includes all the regions that are reset by the evaluation of `e`. When extracting an expression `let x = e1 in e2`, the regions of `x` that escape from `e2` either are reset by `e2`, or appear in its return type. We check that the set of regions that appear in `x` has no intersection with the set of regions that may escape. If there is an intersection, the extraction fails.

Records

WhyML records can generally be extracted to C structs without issues, so long as the type of each field can be extracted to C. Nested records are currently not supported, although it seems possible to do so when the inner one is immutable (when the nested record is mutable, special treatment is needed to avoid memory leaks). Records with mutable fields are boxed in the exact same way as references. Figure 3.15 shows an example of a WhyML program that uses records and its extraction to C.

When a record type has no mutable fields, there is no need to box it when passing it to a function, as the function is guaranteed to not write into it. Therefore, we perform a small transformation, and simply pass the struct by value in the extracted code. This makes the extracted code a bit clearer. However, it is not clear that it is an optimization in terms of performance, especially for large structs.

3.4.8 Arrays

WhyML does not have a built-in array type, but the standard library features several array structures which are used in many WhyML programs. Moreover, many C programs use stack-allocated arrays, and we need some way to transcribe them into WhyML programs. These two constructs are not exactly analogous, but seem like a natural fit nonetheless. The main difficulty is that arrays are not a built-in WhyML type, but a user type. The extraction mechanism has no great way to infer that it should be extracted to an array rather than a struct. I ended up using an ad-hoc approach. Figure 3.16 is an excerpt of the definition of arrays indexed by 32-bit integers in Why3's standard library.

Note that the array elements are actually stored in a ghost field (an infinite map from integers to the type of the elements). This is really a model of ar-

```
type array [@extraction:array] ’a = private {
  mutable ghost elts : int → ’a;
                length : int32;
} invariant { 0 ≤ length }

val make [@extraction:array_make] (n: int32) (v: ’a) : array ’a
  requires { n ≥ 0 }
  ensures  { forall i:int. 0 ≤ i < n → result[i] = v }
  ensures  { result.length = n }

val ([]) (a: array ’a) (i: int32) : ’a
  requires { 0 ≤ i < a.length }
  ensures  { result = a[i] }

val ([]←) (a: array ’a) (i: int32) (v: ’a) : unit writes {a}
  requires { 0 ≤ i < a.length }
  ensures  { a.elts = M.set (old a.elts) i v }
```

Figure 3.16: Definition of 32-bit arrays

rays, analogous to the C memory models from Sec. 3.3. The extraction driver can specify that `([])` and `([]<-)` are replaced by array access and assignment respectively. The type declaration and the `make` function are dealt with by the extraction mechanism in an ad-hoc way. The `[@extraction:array]` attribute in the type definition specifies that we are declaring an array type. The `[@extraction:array_make]` attribute specifies that the `make` function should be replaced by an array declaration, followed by an initialization loop. For example, the expression `let t = make 5 (42:int32) in e` would be extracted as:

```
{
  int32_t t[5];
  int i;
  for (i = 0; i < 5; ++i) {
    t[i] = 42;
  }
  ⟦e⟧
}
```

There are a few caveats. First, the first argument of `make` (size of the array) needs to be a constant expression. We detect this during extraction in a syntactic way. Second, we check that the address of `t` (which resides on the stack in the extracted C program) does not escape its scope, using the same approach as for references and records with mutable fields.

3.4.9 Extracting a multi-file library

The extraction mechanism has a number of useful features to extract libraries comprised of multiple C files. The extraction command takes a Why3 module as argument. If the corresponding flags are set, it extracts this module and all of the other modules it depends on. For each module, a C file and a header file are generated. The header file contains the declarations of all types and functions in the module, as well as `include` directives for all the headers of the dependencies. Each C file includes its own header file.

The extraction drivers also feature directives that help with multiple-file extraction. The `interface` directive is similar to the `prelude` directive, but prints its content in the header file rather than the source file. Finally, the `prelude export` directive prints its content in all C files that depend on the current module. For example, the default driver for C extraction uses this directive to include `stdint.h` in all files that use the type `int32` and in their interfaces. Finally, the trouble with this recursive extraction scheme is that it attempts to extract files from the WhyML standard library. These files typically do not contain anything relevant for extraction, as all they do is declare type and functions that are only defined in the driver. They end up cluttering the extracted code with empty files and useless includes. The `remove module` driver directive mitigates this by explicitly excluding them from the extraction. Below is a representative sample of the section of the default C extraction driver that deals with the `Int32` standard library module.

```
module mach.int.Int32

  prelude export "#include <stdint.h>"
  interface export "#include <stdint.h>"

  syntax type int32 "int32_t"

  syntax val (+)     "%1 + %2"   prec 4 4 3
  ...

  remove module

end
```

Let us look at a complete example of extraction of a simple multi-file library. It involves two WhyML modules, as well as the `Int32` module from the standard library. The WhyML source files are reproduced in Fig. 3.17. The driver is the default C extraction driver. All relevant directives are in the excerpt above. After extraction, we obtain two C source files and two header files. They are reproduced in Fig. 3.18.

```
module A

use mach.int.Int32

let f (x:int32) = x + x

let h (x:int32) = x + 3

end
```

(a) a.mlw

```
module B

use mach.int.Int32
use a.A

let g (x:int32) = f (x + 4)

end
```

(b) b.mlw

Figure 3.17: Source files of a simple library

3.4.10 Making extracted code readable

A number of program transformations are performed by the extraction. The end goal is to make the extracted code as readable as possible. This is correlated,

```c
#include "a.h"
#include <stdint.h>

int32_t f(int32_t x) {
  return x + x;
}

int32_t h(int32_t x) {
  return x + 3;
}
```

(a) a.c

```c
#include "b.h"
#include <stdint.h>

int32_t g(int32_t x) {
  return f(x + 4);
}
```

(b) b.c

```c
#ifndef A_H_INCLUDED

#include <stdint.h>

int32_t f(int32_t x);

int32_t h(int32_t x);

#define A_H_INCLUDED
#endif // A_H_INCLUDED
```

(c) a.h

```c
#ifndef B_H_INCLUDED

#include "a.h"
#include <stdint.h>

int32_t g(int32_t x);

#define B_H_INCLUDED
#endif // B_H_INCLUDED
```

(d) b.h

Figure 3.18: Extracted code for our simple library.

but not always equivalent to making it shorter. This section discusses a few of these transformations.

Proxy variable elimination

As explained in Sec. 2.1, Why3 performs some normalizations on user programs. This makes verification conditions easier to compute. As a result, ML programs are in A-normal form, that is, all function arguments are trivial. This means that programs contain many variables that are used only once. For example, the expression `f (a+b) (c+d)` would be normalized to something like `let o = c+d in let o1 = a+b in f o1 o` (recall that function arguments are evaluated from right to left). As a result, the extracted code is far longer than the original user code, and harder to read. I have added a transformation pass right after the compilation from WhyML to ML. The proxy variables that are added by the A-normalization are marked internally. The transformation reverts the A-normalization by inlining the variables (that is, replacing them by their definition).

Not all proxy variables can safely be inlined this way. The nature of the A-normal form ensures that each proxy variable is used only once, so inlining their definition introduces no complexity penalty (we are not duplicating computations). The issue comes from the order of the side effects in the computations. Take the example of the function call `f e1 e2`, and suppose that the evaluations of the expressions `e1` and `e2` both have side effects. WhyML enforces that `e2` is computed before `e1`, and makes sure of it by properly ordering the proxy variables. The C standard does not specify an evaluation order for function arguments. Therefore, it would not be correct to inline the computations of `e1` and `e2`, because they might be computed in the wrong order.

In the end, proxy variables are only inlined when their computation has no side effects. We could be a little cleverer and allow inlining in some situations even with side effects, such as for the first function argument (which should be computed last anyway). However, the current heuristic is simple and already greatly reduces the number of extra variables in the extracted code.

Parentheses elimination

The precedence system discussed in Sec. 3.4.4 is expressive enough to remove almost all the parentheses that are not needed. However, this is not necessarily what we want to do. For example, the precedences of the bitwise operators `&`, `^`, `|`, `&&`, and `||` are strictly ordered. This means that the expression `x && y | z` is unambiguously equivalent to `x && (y | z)`, not `(x && y) | z`. Unfortunately, no one remembers the precedences of these operators. It is also not particularly intuitive that the bitwise “or” operator has higher priority than the boolean “and” operator. In fact, GCC emits the `-Wparentheses` warning for expressions such as this one. In such cases, adding “extraneous” parentheses makes the code more readable, not less. Fortunately, this is easy enough to do. In the default C extraction driver, the precedences of the boolean operators are as follows.

```
syntax val orb  "%1 || %2" prec 12 5 5
syntax val andb "%1 && %2" prec 11 5 5
syntax val xorb "%1 ^ %2" prec 9 5 5
```

The strictest possible rule for `||`, for example, would be `prec 12 12 11`, which would allow any boolean or bitwise operator to take priority over `||` without parentheses. We require a precedence level of 5 or less (essentially that of arithmetic expressions) to avoid parentheses. Essentially all complex boolean and bitwise expressions are parenthesized.

Explicit type conversions

In WhyML programs, the only way to cast a bounded integer to another bounded integer type is through a function. We declare a `val` function such as `to_int64` below, give it the appropriate specification, and replace it by a cast in the driver.

```
val to_int64 (x:int32) : int64
  ensures { int64'int result = int32'int x }

syntax val to_int64 "(int64_t)%1"
```

In C programs, some casts may be omitted. Typically, when using an integer in a context where a wider integer type is required (i.e., there is no risk of information loss), the conversion occurs implicitly. We could imagine detecting some of these cases in the extraction mechanism and deleting the gratuitous type conversions. This would shorten the extracted code. However, it tends to hurt code readability more than it helps. Detecting which type conversions can be omitted is also not trivial. Consider the WhyML expression `lsl (to_int64 x) 50`, where `lsl` models the C left logical shift operator `«` and `x` is a 32-bit integer. The expression `(int64_t)x « 50` is perfectly valid, but `x « 50` invokes undefined behavior. I ended up not implementing this transformation.

Expression simplifications

Extraction sometimes generates code patterns that should never appear in handwritten C programs. For example, the fact that booleans are represented as integers in C programs, but not in WhyML programs, sometimes leads WhyML programmers to write expressions such as `if (x <> 0) ...` or `if x then 0 else 1`. A small number of simplifications have been implemented. For example, the patterns above are replaced by `if (!x) ...` and `!x` respectively. Other examples include purely syntactic transformations, such as simplifying `(*x).a` into `x->a`, or `!(a==b)` into `a != b`.

Many complex optimizations could be implemented. However, this would introduce yet another source of bugs in the untrusted extraction code. In an effort to keep the extraction code as simple as possible, I implemented only very small, local transformations that increase code readability.

3.4.11 Contiki's ring buffer, extracted back to C

Let us now evaluate the extraction on our running example, Contiki's ring buffer. Figure 3.19 shows a side-by-side comparison of Contiki's original code, and the code that is extracted from our WhyML implementation. Whitespace has been minimally edited for readability. There are two main differences between the two programs.

The first difference is that Contiki functions return values of type `int`, while our WhyML implementation uses `int32_t`. Indeed, as explained in Sec. 3.2.2, we have made the assumption that `int` is 32 bits wide. In order to make this assumption clear in the extracted code, we use the fixed-width `int32_t` type.

The second visible difference is the presence of type conversions between `uint8_t` and `int32_t`. Indeed, the original code performs many implicit type conversions. Many are hidden from the extracted code, due to our custom `uint8` operations described at the beginning of Sec. 3.2.2. The type conversions to `uint8_t` in `ringbuf_put` and `ringbuf_get` are also implicitly performed by the original code. They could likely be removed from the extracted code using similar methods, but I am not convinced it would be an improvement. Indeed, they make the fact that the arithmetic computations are done on values of type `int`, and not `uint8_t`, which is not obvious at first glance.

The type conversions in pointer lookups and affectations are artifacts from our memory model. Indeed, the `get_ofs` function takes an argument of type `int32` in our memory model, so we had to add casts in the WhyML implementation. This could be avoided by adding various versions of `get_ofs` to the memory model, one for each integer type. They would all have the same specification. The main reason why this was not done is out of a concern to avoid code duplication, especially with a model still under active development.

Overall, the extracted code is clearly similar to the original code. They can be compared line by line, and there are only a handful of differences. Moreover, most of the differences, such as the remaining type conversions, arguably make the extracted code clearer than the original. In this example, the extraction managed to output idiomatic code that could be reviewed by a human. Despite the fact that we have merely produced a correct program rather than verifying the original code, the extracted code is similar enough to Contiki's implementation to serve as evidence of the latter's correctness.

```
struct ringbuf {
  uint8_t *data;
  uint8_t mask;
  uint8_t put_ptr, get_ptr;
};

void ringbuf_init(struct ringbuf *r, uint8_t
     *dataptr, uint8_t size) {
  r->data = dataptr;
  r->mask = size - 1;
  r->put_ptr = 0;
  r->get_ptr = 0;
}

int ringbuf_size(struct ringbuf *r) {
  return r->mask + 1;
}

int ringbuf_elements(struct ringbuf *r) {
  return (r->put_ptr - r->get_ptr) & r->mask;
}

int ringbuf_put(struct ringbuf *r, uint8_t c)
      {
  if(((r->put_ptr - r->get_ptr) & r->mask) ==
        r->mask) {
    return 0;
  }
  r->data[r->put_ptr] = c;
  r->put_ptr = (r->put_ptr + 1) & r->mask;
  return 1;
}

int ringbuf_get(struct ringbuf *r) {
  uint8_t c;
  if(((r->put_ptr - r->get_ptr) & r->mask) >
       0) {
    c = r->data[r->get_ptr];
    r->get_ptr = (r->get_ptr + 1) & r->mask;
    return c;
  } else {
    return -1;
  }
}
```

(a) Contiki's ring buffer.

```
struct ringbuf {
  uint8_t * data;
  uint8_t mask;
  uint8_t put_ptr;
  uint8_t get_ptr;
};

void ringbuf_init(struct ringbuf * r, uint8_t
      * dataptr, uint8_t size) {
  r->data = dataptr;
  r->mask = (uint8_t)(size - 1);
  r->put_ptr = 0;
  r->get_ptr = 0;
}

int32_t ringbuf_size(struct ringbuf * r) {
  return r->mask + 1;
}

int32_t ringbuf_elements(struct ringbuf * r)
     {
  return (r->put_ptr - r->get_ptr) & r->mask;
}

int32_t ringbuf_put(struct ringbuf * r,
     uint8_t c) {
  if (((r->put_ptr - r->get_ptr) & r->mask)
       == (int32_t)r->mask) {
    return 0;
  }
  r->data[(int32_t)r->put_ptr] = c;
  r->put_ptr = (uint8_t)((r->put_ptr + 1) & r
       ->mask);
  return 1;
}

int32_t ringbuf_get(struct ringbuf * r) {
  uint8_t c;
  if (((r->put_ptr - r->get_ptr) & r->mask) >
        0) {
    c = r->data[(int32_t)r->get_ptr];
    r->get_ptr = (uint8_t)((r->get_ptr + 1) &
         r->mask);
    return (int32_t)c;
  } else {
    return -1;
  }
}
```

(b) Extracted code.

Figure 3.19: Extraction of Contiki's ring buffer: a side-by-side comparison.

3.5 State of the art, conclusion

This chapter introduced two main contributions: a model of the C language in WhyML, and a mechanism to translate WhyML programs into C programs. Together, they allow Why3 users to verify the functional correctness of C programs using the following approach. First, re-implement the C program in WhyML using the memory model. Second, verify the program with Why3 using any of the usual methods. Third, use the extraction mechanism to generate correct-by-construction C code. The goal is for the end result to be close enough to the original code to be usable as a drop-in replacement for it. In this chapter, I illustrated the approach with a small case study: a data structure from the Contiki operating system and its API. The extracted code ended up extremely similar to the original. Let us now review related work, evaluate our design choices, and consider possible future work.

3.5.1 Related work

Let us go over a few other tools that enable the deductive verification of C programs. One family of such tools uses the Frama-C and Why3 platforms in combination. The first of these tools was the Caduceus tool [36], which translated C programs annotated in the ACSL specification language [8] into the functional language of the Why platform, which computed verification conditions and interfaced with automated solvers in much the same way as its successor Why3 does. An example of proof performed with Caduceus is the verification of the Schorr-Waite algorithm by Hubert and Marché [52]. After Frama-C was developed, Caduceus was replaced by the Jessie plugin [67] for the Frama-C environment [25], leveraging the latter's C syntax and type checker. Jessie translates the C programs and ACSL annotations to Why3 (originally to Why), which computes the verification conditions and interfaces with provers as usual. Finally, the more recent WP plugin for Frama-C [7] computes its own verification conditions within Frama-C using built-in memory models. Why3 is used only as a way to interact with the theorem provers. Examples of programs verified with Frama-C/WP include various Contiki modules [14, 80]. When comparing these tools with my extraction-based approach, the main tradeoff is the one we outlined in the introduction. The tools based on the Frama-C platform verify the original source code, without needing to modify it outside of the addition of specification-oriented comments. On the other hand, my approach requires re-implementing the algorithms from scratch, and compiling them back to C, resulting in the verification of a closely related but not quite equivalent program. However, using WhyML as the source language makes it much easier to leverage Why3's theorem-proving strengths. As the next chapter shows, the GMP algorithms that I verified are intricate enough that I do not believe they can currently be verified using the WP plugin in a similar time frame.

Appel's Verified Software Toolchain [6] also analyzes C program at the source level. More precisely, it targets *C minor*, an intermediate representation from the certified C compiler CompCert [63]. It uses a program logic based on concurrent separation logic, with a notion of permissions. A series of unverified, static analyzers implemented in the Coq theorem prover output assertions about the program, which are then checked by a verified core analyzer. The use of Compcert allows the verification to go all the way down to the machine language.

As a result, the trusted code base of the compiled programs is very small. As an example, the approach has been used to verify that a concurrent messaging system is race-free, memory safe and functionally correct [66].

The approach of "verifying" C programs by extracting them from a verified implementation in a high-level language is not new to this work. Programs written in the functional language of the PVS theorem prover can also be compiled to C using the PVS2C code generator [32, 90]. In comparison to our approach, the source programs are much less similar to C programs. For example, users are not required to manage the memory manually using an embedding of the C memory in the functional language. Instead, the translation to C uses reference counting to free unused variables and use in-place updates wherever possible. As a result, the user does not appear to have as much control over the compiled code as in our approach, as the translation process is not as straightforward.

Another family of tools for the verification of C programs interact with the Isabelle/HOL theorem prover [78]. The seL4 verified microkernel [56] was famously verified in Isabelle through successive manual proofs of refinement between the C implementation, executable specification (generated from a Haskell prototype), and a high-level abstract specification. Several tools have since been developed in order to automate these refinement proofs. Greenaway's AutoCorres tool [46, 47] automatically performs a sequence of abstractions from a shallow embedding of C in Isabelle to higher-level specifications more suitable to human reasoning, and generates proofs of correspondence for each of these translations. The Cogent compiler [5] translates programs written in a high-level language (tailored to writing systems code) into C. It is built on top of AutoCorres and generates a formal specification of the compiled code in Isabelle/HOL, as well as a refinement proof that the generated C code correctly implement this semantics. The authors have used Cogent to verify two file systems. In comparison to our approach, Cogent seems to produce code that is not as efficient (the authors report around a 100% slowdown between the original C file systems and the generated versions). However, correctness proofs are automatically generated at each translation step, which is not at all the case of our approach.

Finally, programs written in the F* verification-oriented programming language [91] can be extracted to efficient C code using the KreMLin tool [81]. Much like in our approach, F* programs are only accepted by KreMLin if they are implemented in a shallow embedding of C in F*, with manual memory management and some restrictions (no higher-order programs or recursive data types). The memory model, called HyperStack, uses a tree-like model of the heap and stack. Regions of the heap or the stack are allocated and managed manually. As a result, HyperStack is more verbose, but also much more expressive than our memory model. The compilation from F* to C has been formalized on paper. It targets Clight, an intermediate representation from the CompCert verified compiler. Much as with our approach, the extracted C code is efficient and easily predictable for the user, as it strongly resembles the source F* code. KreMLin has been used to develop the HACL* library [96], which provides numerous cryptographic primitives and is as efficient as the non-verified alternatives.

3.5.2 Memory model evaluation

When verifying programs with potential aliasing, one of the top issues is the frame problem. When one memory zone is modified, the model should be able to describe which memory areas did or did not change. My memory model does so using Why3's built-in alias tracking. Other solutions exist. For example, one could embed some fragment of separation logic [83], or use a more structured memory model, like F*'s hyper-heaps.

The choice of relying on Why3's alias tracking is an economical one. It works well enough in a wide range of situations, and does not require a very engineered model. As a result, the core memory model is only about 200 lines long. The main downside is that it is not expressive enough to model all C programs. Many alias-heavy programs effectively cannot be implemented using my model. I have developed ad-hoc tools to increase my model's expressivity, such as the ones shown in Sec. 3.3.3 and Sec. 3.3.4, but they are not sufficient. Furthermore, the more such hacks we introduce, the more we stretch the trust that the user has to place in the model. In the cases that are complex enough to warrant even more hacks, it seems that we would be better off using a different model.

Some features are still missing from my model. Some of them are absent by design, such as pointer arithmetic and the address operator `&`. However, there are many features that could be added to the model, but have not been yet. Indeed, the development of my model was parallel to and largely driven by my case study, the proof of a fragment of the GMP library. Therefore, many things that my model could in theory support were not added because they were not needed for my case study, and I did not have enough time to work on them. For example, pointer subtraction inside a memory block could be implemented. We could check that two pointers are inside the same memory block using their `zone` fields, and perform arithmetic on their `offset` fields.

The model also does not check that memory reads are performed only on initialized areas. I experimented with this a little bit. One could imagine adding to pointers a ghost array of booleans that specifies whether each memory cell is initialized, for example. However, this would require quite a bit of extra work in program specifications and in proofs. Moreover, bugs related to uninitialized reads tend to be caught anyway when writing WhyML programs, as it tends to be very hard in practice to prove function preconditions on values read from uninitialized memory areas. However, it is still possible to write a WhyML program that performs an unitialized read and does something pointless with it, e.g., `x - x`. Such a program would invoke undefined behavior when extracted to C.

There are also some portability issues. Some driver directives use GCC built-ins such as `__builtin_clzll`, so they might not work with other C compilers. My model also requires pointer sizes to have type `int32`, and assumes that the `int` type is 32 bits wide and the `char` type is 8 bits wide. All these hypotheses are somewhat arbitrary. In retrospect, I regret not introducing more genericity from the start. For example, using Why3's module cloning feature, which is analogous to functor instantiation in OCaml, we could generate multiple versions of the memory model and vary the various types involved. It would be even better if we allowed programs to use various types as pointer sizes and offset inside the same function, as is permitted in C.

A more complex issue that my model tackles poorly is the stack. My model features a specification for `alloca` and minimal support, but it is unsafe. It has no good way to ensure that stack-allocated variables do not escape from their scope (this can be caught at extraction, but not during the proof of the WhyML program itself). Similarly, nothing prevents a WhyML user from allocating a pointer on the stack and calling `free` on it. There are a few potential ways to handle the stack. For example, we could track it more explicitly, with a global variable representing the state of the stack. The main invariant of the C stack is that at the end of each function call, the stack pointer is back to the same position as it was at the beginning of the function. But there is currently no good way to specify this invariant in Why3 in a global way. Type invariants are too restrictive for this. We could manually add a clause that says `stack = old stack` to the specification of all functions, but this is somewhat unpleasant. I expect this problem to be hard but solvable, potentially with some modifications to Why3. The main reason why this did not happen is that again, it was not needed for my case study.

In the end, I am mostly happy with my choice of relying on Why3's alias tracking for the memory model. In the simpler cases (in terms of aliasing), the model is quite lean and justifies the choice of Why3 as a verification platform, as it requires very little extra work on the part of the user in proofs and specifications.

However, in the more complex cases, the type system is too constraining. The user needs to either hack it with increasingly complex abstract functions (as I do in Sec. 3.3.4), or use a memory model that explicitly does not involve regions, so as to evade Why3's alias tracking. However, in this last case, it is not so easy to have mutability without involving regions. The memory model often ends up convoluted, and it becomes even harder than usual to justify that it is correct. An example of this is the `mpz` memory model in Sec. 4.6.1.

Developing a more expressive memory model would not be an easy task. One of the reasons is that, while Why3's type system models side effects in a rich way, many effects cannot be specified in the surface language. For example, the reset effect occurs as a result of user writes (either in the code or in a specification), but one cannot specify explicitly that a region is reset. This is why some of my specifications use somewhat odd `writes` clauses that are only there to trigger the reset effect. As part of this work, I made some contributions to Why3 that help with this somewhat, such as the `partial` keyword. More notably, the `alias` keyword, with which users can specify in a `val` declaration that two regions are always aliased, was added during this work, and is extremely important to my model. It is not my contribution, although I claim credit for pestering the developers until they added it.

3.5.3 Correctness, trusted code base

Why3's trusted code base is quite extensive. In order to have confidence in programs proved by Why3, the user first needs to trust that Why3 computes the right verification conditions. Second, they need to trust that they were correctly translated by Why3 into the input language of the various automated solvers involved, and that the automated solvers themselves are correct. Any Why3 user already needs to trust a rather large code base.

This works aims at using Why3 to verify C programs. To do so, we add two components to the trusted code base. First, we model the C language in WhyML using what is essentially a large set of axioms. Reviewing these axioms is not so easy. They total a few hundred lines of WhyML code. However, their correctness relies not only on the C standard, but also on our extraction drivers, as well as specific inner workings of Why3's region system.

Second, we translate WhyML programs back to C, using extraction code that is not verified. In order to increase confidence in the extraction, we would like it to be as simple as possible. As a result, the translation from WhyML to C is mostly syntactic, and many WhyML features are not supported by the extraction. However, compromises need to be made between the simplicity of the extraction and the ability to produce a wide variety of C programs. It is easy to make the decision of not supporting WhyML features that are not idiomatic C features, such as automatic memory management or higher order functions. However, if too many C features cannot be expressed in WhyML and translated back to C, then our approach is unusable in practice. In order to support non-trivial use cases, I ended up adding a number of non-trivial features to the C extraction, and it would no longer be entirely fair to claim that it is simple enough to trust after a simple code review.

The question of the correctness of the memory model and extraction goes further than fixing a few bugs in the extraction code. This approach to verifying C programs by extraction has proved reasonably effective for my case study. However, not much theoretical work has yet been done to justify that it is sound, that is, that the translation to C preserves the semantics of WhyML programs.

We could imagine a proof that links the semantics of WhyML programs to the semantics of the extracted program through some bisimulation relation. Formalized semantics of the C language or related intermediate languages exist in the wild. For example, the CompCert compiler [63] compiles C to x86 assembly through ten intermediate languages, which are all given formal semantics. They are used by other pieces of related work. For example, the formalization of KreMLin's compilation from F* to C targets CompCert's Clight intermediate language [81]. However, no formalized semantics of the WhyML language is conveniently available yet. We would need to develop one first. The formalization would also need to take the extraction drivers into account, in order to be able to state and verify the soundness of the memory model. Given the amount of work required, I do not particularly regret not having worked in this direction yet. It would not have been realistic to do this in a timely manner while also working on a sizable use case as a proof of concept. The most direct point of comparison is probably the KreMLin formalization, and it is a 90-page paper with more than ten authors. Nonetheless, formalizing the correctness of the memory model and extraction to C will need to be adressed in future work. Until then, the proof is only in the pudding.

Chapter 4

WhyMP

Using the approach enabled by the tools presented in last chapter, I have verified a fragment of the GMP library. Through Why3's extraction mechanism, we obtain an efficient and formally-verified C library called WhyMP [72]. To the best of my knowledge, this work is both the first formally verified comprehensive state-of-the-art arbitrary-precision integer library, and the largest WhyML development yet.

This chapter presents WhyMP's algorithms and outlines their proofs. There are two kinds of facts to prove when verifying arithmetic algorithms. First, there are the properties related to safety, such as integer overflow checks or array bounds checks. Most of these facts are proved automatically by automated solvers with very little user input. This leaves the higher-level mathematical facts about functional correctness as the most difficult part of the proofs by far. The Why3 proofs of these facts tend to be made up of long assertions that are not unlike paper proofs. In the end, showing the WhyML code would not be very legible. So, the proofs we present in this chapter deal with faithful transcriptions of the algorithms in detailed pseudocode. We give the specifications and invariants, and provide step-by-step explanations of the algorithms and proofs that their implementations match the specifications.

GMP has multiple layers, which handle different types of numbers (natural, relative, rational, floating-point) through various layers of abstraction. This work focuses mostly on the `mpn` layer, which handles natural numbers. There is also some support for the `mpz` layer, which handles relative numbers, although I have only verified a relatively small number of `mpz` algorithms. Note that I did not verify the entire `mpn` layer either. Indeed, for each operation, `mpn` implements many different algorithms, each most suitable for different input sizes. The fragment of `mpn` that I have verified includes at least one algorithm for each of addition, subtraction, multiplication, division, square root, and fast modular exponentiation. Most of the functional correctness proofs in this chapter are largely drawn from a previous paper in which I present the `mpn` algorithms [84], although the sections on `mpz` and the most recent `mpn` proofs (modular exponentiation, base conversions) are original.

The chapter is structured as follows. We first go over GMP's number representation and its modeling in WhyML (Sec. 4.1). Section 4.2 regroups the proofs of GMP's so-called "schoolbook" algorithms. These algorithms for the four basic operations have the same structure as the ones taught in school, although they have been optimized in various, more or less complex ways. This is especially the case for the division function, which uses an intricate 3-by-2 division primitive. GMP's fast modular exponentiation algorithm is described in Sec. 4.3. It required formalizing various concepts of modular arithmetic, such as computations in Montgomery form. In Sec. 4.4, we describe one family of GMP's divide-and-conquer multiplication algorithms that involve intricate buffer manipulation and carry propagation. Section 4.5 describes another divide-and-conquer algorithm, which computes the square root of a large integer. However, the most interesting part of that algorithm is the base case, which computes the square root of a machine integer using an intricate implementation of Newton's method in fixed-point arithmetic. Most of the work was spent formalizing concepts of fixed-point arithmetic in Why3. Section 4.6 describes my formalization of the `mpz` layer. Its algorithms are much more permissive than those of the `mpn` layer in terms of parameter aliasing, which required a custom memory model. GMP's base conversion algorithms are described in Sec. 4.7. They convert back and forth between user-readable character strings and GMP's internal number representation. The main verification challenge was the modeling of the various required concepts: character strings, the ASCII character encoding, and so on. Section 4.8 compares WhyMP and GMP in terms of compatibility and performance benchmarks. Finally, Sec. 4.9 goes over lessons learned from the development of WhyMP, some related work, and some possible lines of future work.

4.1 Modeling GMP inside Why3

This section presents GMP's representation of large integers as arrays of full words. We then present an example of our workflow on a simple GMP routine, GMP's integer copy function.

4.1.1 Integer representation

In GMP, natural integers are represented as arrays of unsigned integers called *limbs*. We set a radix $\beta = 2^{64}$ (also called `radix` in the formal development). Any natural number N has a unique decomposition $\sum_{k=0}^{n-1} a[k]\beta^k$ in base β such that $a[n-1] \neq 0$, and is represented as the buffer $a[0]a[1]\dots a[n-1]$ (with the least significant limb first). If $N = 0$, it is represented by a 0-length array.

For the sake of efficiency, there is no memory management in GMP's low-level functions. The caller code has to keep track of number sizes. Function operands are specified by two separate parameters: a pointer to their least significant limb and a limb count. In our case, since $\beta = 2^{64}$, each limb lies between 0 and $2^{64} - 1$, so the type `uint64` of unsigned 64-bit integers is used to represent limbs.

```
type limb = uint64

type t = ptr limb
```

Let us now establish the link between mathematical integers and arrays of machine integers. If a pointer a is valid over a size n, we denote $\texttt{value}(a, n) = \overline{a[0]\dots a[n-1]} = \sum_{k=0}^{n-1} a[k]\beta^k$. We also use the same notation for the value of a slice of the pointer that does not start at 0. For example, we denote value(incr$(a,k),p)$ as $\overline{a[k]\dots a[k+p-1]}$. The value function is defined as follows in WhyML:

```
let rec ghost function value_sub (x:map int limb) (n:int) (m:int) : int
  variant { m - n }
=
  if n < m
  then to_int x[n] + radix * value_sub x (n+1) m
  else 0

function value (x:ptr limb) (sz:int) : int =
  value_sub x.data.elts x.offset (x.offset + sz)
```

Notice that the return type of value is the type int of unbounded mathematical integers. This type cannot be used in programs meant to be extracted to C, but the value function cannot either (it is a purely logical function). Therefore, its result will never interact with the program code, so we may as well choose the type that makes formal verification easiest, which is int.

The following lemmas express what happens to the value of a big integer when part of it is modified. In particular, loops tend to change only one end of a big integer (usually by increasing its length), and being able to separate what changed from what did not is crucial to prove that loop invariants are preserved.

Lemma 1. *Let p a pointer valid over the length n, and let $0 \leq k < n$.*

$$\overline{p[0]\dots p[n-1]} = \overline{p[0]\dots p[k-1]} + \beta^k \overline{p[k]\dots p[n-1]} \qquad [\texttt{value\_sub\_concat}]$$

$$\overline{p[0]\dots p[n-1]} = \overline{p[0]\dots p[n-2]} + \beta^{n-1} p[n-1] \qquad [\texttt{value\_sub\_tail}]$$

$$\overline{p[0]\dots p[n-1]} = p[0] + \beta \overline{p[1]\dots p[n-1]} \qquad [\texttt{value\_sub\_head}]$$

$$\overline{p[0]\dots p[k-1]\ v\ p[k+1]\dots p[n-1]}$$
$$= \overline{p[0]\dots p[k]\dots p[n-1]} + \beta^k (v - p[k]) \qquad [\texttt{value\_sub\_update}]$$

We also need some bounds for the value of an integer of size n. These follow easily from the fact that the values $a[i]$ lie between 0 and $\beta - 1$.

Lemma 2. *Let p a pointer valid over the size n.*

$$0 \leq \overline{p[0]\dots p[n-1]} \qquad [\texttt{value\_sub\_lower\_bound}]$$

$$p[n-1]\beta^{n-1} \leq \overline{p[0]\dots p[n-1]} \qquad [\texttt{value\_sub\_lower\_bound\_tight}]$$

$$\overline{p[0]\dots p[n-1]} < \beta^n \qquad [\texttt{value\_sub\_upper\_bound}]$$

$$\overline{p[0]\dots p[n-1]} < (p[n-1]+1)\beta^{n-1} \qquad [\texttt{value\_sub\_upper\_bound\_tight}]$$

4.1.2 Example GMP function: mpn_copyd

Let us now go over a simple GMP routine as an example. The mpn_copyd function is shown in Fig. 4.1. It takes two pointers rp and up valid over a size n and copies the limbs pointed by up into rp in decreasing order, that is, starting with the most significant limbs. (Another copy function, mpn_copyi, does the same thing in increasing order.) Note that up and rp are allowed to point to zones longer than n, or even to point to the middle of a number. The only length requirement is that there are at least n valid limbs to the right of each pointer.

```
void mpn_copyd (mp_ptr rp, mp_srcptr up, mp_size_t n)
{
  mp_size_t i;
  for (i = n - 1; i >= 0; i--)
    rp[i] = up[i];
}
```

```
let wmpn_copyd (rp up:t) (n:int32) : unit
  requires { valid up n ∧ valid rp n }
  ensures { forall i. 0 ≤ i < n → rp[offset rp + n] = up[offset up + n] }
  ensures { forall j. (j < offset rp ∨ offset rp + n ≤ j)
                     → rp[j] = old rp[j] }
=
  for i = n-1 downto 0 do
    invariant { forall j. i + 1 ≤ j < n →
                       rp[offset rp + j] = up[offset up + j] }
    invariant { forall j. (j < offset rp ∨ offset rp + n ≤ j) →
                       rp[j] = old rp[j] }
    let lu = C.get_ofs up i in
    C.set_ofs rp i lu
  done
```

Figure 4.1: GMP's copy function and its WhyML transcription.

There is an additional restriction related to pointer aliasing. If the pointers rp and up overlap, the function still behaves properly on the condition that rp $\geq$ up (otherwise, the copy would overwrite parts of up that have not been copied yet). This behavior is not documented, but GMP makes heavy use of it nonetheless. In that case, the contents of up are of course modified. This is one of the cases where my C memory model is more restrictive than we would like. The WhyML transcription as written on Fig. 4.1 cannot be used on two pointers that overlap (or even point to the same memory zone while being separated). On the plus side, the specification is much simpler than it would otherwise be. There is no need to specify the changes that may occur in up (or even the fact that it does not change). However, in order to copy between pointers inside the same block, we need to do extra work. Details on this extra work can be found in Sec. 3.3.4. In the rest of the section, let us simply assume that rp and up are separated, and that an extra copy function handles the remaining cases.

Let us briefly compare the WhyML transcription of mpn_copyd to the original C code. The function body is extremely similar, as expected for such a simple function. The only thing worth mentioning is the choice of the type used to model mp_size_t. In GMP, this type is usually either int32 or int64 depending on the system architecture. We choose int32 somewhat arbitrarily. It would be desirable to introduce some kind of genericity in the WhyML implementation, so as to prove that the programs are valid for both choices. However, I have not found a great way to do this while preserving a good degree of proof automation.

The specification and proof of wmpn_copyd are relatively simple. The only precondition is that both input pointers are valid over the length n. Note that there is no precondition requiring n to be non-negative. The function can be called on a negative n argument, in which case it does nothing and the specification also proves nothing. As explained in Chap. 2, the specification says nothing about aliasing, as the constraints discussed above are implicitly enforced by Why3's type system rather than encoded in the logic. The first postcondition states that the first n cells of `rp` are now a copy of the first n cells of `up`. Note that nothing requires `rp` or `up` to point to the start of a memory block, or n to encompass the whole block. Therefore, the second postcondition is necessary. It states that the rest of `rp` is left unchanged. The body of the function is a straightforward loop. Much as one could expect, the loop invariants simply express the same properties as the postcondition on a section of the array. In this case, no extra work is required from the user. The automated solvers are able to easily prove that the loop invariants are valid and that the postconditions follow from them.

```
void wmpn_copyd(uint64_t * rp, uint64_t * up, int32_t n) {
  int32_t i, o;
  uint64_t lu;
  o = n - 1;
  for (i = o; i >= 0; --i) {
    lu = up[i];
    rp[i] = lu;
  }
}
```

Figure 4.2: Extracted code for `wmpn_copyd`.

The extracted code for `wmpn_copyd` is reproduced in Fig. 4.2. As expected, it is extremely similar to the original GMP code. It is a little bit less compact, as the extra variables `o` (added by Why3 as part of the A-normalization process, see Sec. 2.1) and `lu` (added by the user for proof purposes) were not eliminated. However, we can expect a sane compiler to produce very similar outputs for both functions.

4.2 Schoolbook algorithms

This section regroups the schoolbook algorithms implemented in WhyMP. Their structures are typically simple loops. They include integer comparison (Sec. 4.2.1), addition and subtraction (Sec. 4.2.2), multiplication (Sec. 4.2.3), and division (Sec. 4.2.4). The "schoolbook" denomination is found in GMP's source code. However, the division algorithm is so heavily optimized that it can hardly be called a schoolbook algorithm anymore.

4.2.1 Comparison

The `mpn` layer of GMP exposes a single comparison function, which compares two integers of same length (Alg. 1). The algorithm is straightforward: it simply iterates both operands until it finds a difference, starting at the most significant limb.

Since the function involves a loop, we must provide a loop invariant. Here, the loop invariant is that both source operands are identical from offsets $i + 1$ (included) to n (excluded).

Algorithm 1 Comparison of two integers of identical length.

```
Require: valid(x, n), valid(y, n)
Ensure: result > 0 ⇔ value(x, n) > value(y, n)
Ensure: result = 0 ⇔ value(x, n) = value(y, n)
Ensure: result < 0 ⇔ value(x, n) < value(y, n)
  function CMP(x, y, n)
      for $i = n - 1$ downto 0 do
          if $x[i] \neq y[i]$ then
              if $x[i] > y[i]$ then return 1
              else return $-1$
      return 0
```

An additional lemma is required to prove the invariant and complete the proof, it simply says that two big integers with equal limbs at all offsets are equal:

Lemma 3 (`value_sub_frame`).
Let a, b valid over lengths greater than v such that for all $u \leq k < v$, $a[k] = b[k]$. Then $\overline{a[u] \ldots a[v-1]} = \overline{b[u] \ldots b[v-1]}$.

The lemma is proved by a straightforward induction, which translates well into a Why3 lemma function as the recursive call takes care of the inductive case:

```
let rec lemma value_sub_frame (a b:map int limb) (u v:int)
  requires { forall i. u ≤ i < v → a[i] = b[i] }
  variant  { v - u }
  ensures  { value_sub a u v = value_sub b u v }
= if u < v then value_sub_frame a b (u+1) v else ()
```

This lemma shows that the numbers are equal if no difference was found by the end of the loop.

4.2.2 Addition, subtraction

We use the schoolbook algorithms for the addition and subtraction of big integers, represented by their decomposition in base β.

We first need to give the specifications of basic operations on limbs. Much like the three multiplication primitives outlined in Sec. 3.1.1, there are three limb addition primitives.

The first primitive `(+)` is the defensive addition: it requires that the sum of the two inputs does not overflow.

```
val (+) (a b:limb) : limb
  requires { to_int a + to_int b ≤ Limb.max }
  ensures  { to_int result = to_int a + to_int b }
```

The second primitive, `add_mod`, has the semantics of the `+` operator on unsigned integers in C: if there is an overflow, the result wraps around. When compiling WhyML programs to C, both `(+)` and `add_mod` are translated to the C operator `+`. The former's postcondition is stronger and more suitable for SMT solvers, as it does not involve a modulo operator. Therefore, using it simplifies the proofs when its (also stronger) precondition is met. The latter captures the full semantics of the addition, so it can be used in the remaining cases.

```
val add_mod (x y:limb) : limb
  ensures { to_int result = mod (to_int x + to_int y) radix }
```

Finally, the third primitive accepts a carry to be added to the other two operands, and outputs both the carry and the (potentially wrapped-around) result of the addition.

```
val add_with_carry (x y:limb) (c:limb) : (r:limb,d:limb)
  requires { 0 ≤ to_int c ≤ 1 }
  ensures  { to_int r + radix * to_int d = to_int x + to_int y + to_int c }
  ensures  { 0 ≤ to_int d ≤ 1 }
```

Similar primitives are used for limb subtraction.

WhyMP implements many variants of addition and subtraction, depending on whether the operation is done in place, the operation may overflow, the operands are known to be of same length, and so on. As a sufficiently generic example, let us examine the general-case addition (Alg. 2). It assumes that the first operand is longer than the second, and returns a carry out.

Algorithm 2 Addition of two integers.

```
Require: $0 \le n \le m$
Require: $\texttt{valid}(a, m), \texttt{valid}(b, n), \texttt{valid}(r, m)$
Ensure: $\texttt{value}(r, m) + \beta^m \cdot \texttt{result} = \texttt{value}(a, m) + \texttt{value}(b, n)$
Ensure: $0 \le \texttt{result} \le 1$
  function ADD$(r, a, m, b, n)$
      $c \leftarrow 0$
      $i \leftarrow 0$
      while $i < n$ do                                    ▷ Add $b$ to $a$.
          $x \leftarrow a[i]$
          $y \leftarrow b[i]$
          $(z, c) \leftarrow$ ADD_WITH_CARRY$(x, y, c)$
          $r[i] \leftarrow z$
          $i \leftarrow i + 1$
      if $c \ne 0$ then        ▷ Keep copying $a$ into $r$ while propagating the carry.
          while $i < m$ do
              $x \leftarrow a[i]$
              $z \leftarrow$ ADD_MOD$(x, 1)$                          ▷ $c = 1$.
              $r[i] \leftarrow z$
              $i \leftarrow i + 1$
              if $z \ne 0$ then              ▷ No overflow: there is no more carry.
                  $c \leftarrow 0$
                  break
      while $i < m$ do                     ▷ No more carry: copy $a$ into $r$.
          $r[i] \leftarrow a[i]$
          $i \leftarrow i + 1$
      return $c$
```

The algorithm is schoolbook addition with a few optimizations. There are three main steps.

The first step is to add together the two operands over the length of the shorter one. This corresponds to the first `while` loop. Its loop invariants are as follows:

1) $\texttt{value}(r,i) + c\beta^i = \texttt{value}(a,i) + \texttt{value}(b,i)$,

2) $0 \leq i \leq n$.

At the end of the first loop, we have $\texttt{value}(r,n) + c\beta^n = \texttt{value}(a,n) + \texttt{value}(b,n)$ and $i = n$. What remains to be done is to copy the last $m - n$ limbs of a into r and propagate the carry.

The second loop copies a into r while propagating the carry. It is skipped if $c = 0$. Its loop invariants are:

1) $\texttt{value}(r,i) + c\beta^i = \texttt{value}(a,i) + \texttt{value}(b,n)$,

2) $n \leq i \leq m$,

3) $i = m \vee c = 1$.

We break out of the loop whenever the carry c becomes 0 (or if we have finished copying a into r, in which case $i = m$). Note that we are only adding the carry to a limb x of a. There is an overflow if and only if $x = \beta - 1$. There is no need to use `add_with_carry`. It is more efficient to instead use `add_mod` and check if the result is 0, in which case there was an overflow. This is relatively unlikely (probability $1/\beta$ if the operands are randomly drawn from a uniform distribution), so the loop typically only runs for zero or one iteration.

Finally, in the third loop (which is skipped if we already have $i = m$), we only have to copy the last limbs of a into r. The loop invariants are:

1) $\texttt{value}(r,i) + c\beta^i = \texttt{value}(a,i) + \texttt{value}(b,n)$,

2) $n \leq i \leq m$,

3) $i = m \vee c = 0$.

At the end of the loop, we can return c and easily see that the postcondition is verified.

In GMP, all addition and subtraction functions can be used in place, simply by passing the same pointer both as the result and as one of the operands. Why3's aliasing constraints make the situation more difficult for us. In the WhyMP proof, I have verified separate, in-place versions of the addition and subtraction functions. I have avoided code and proof duplication using the techniques from Sec. 3.3.4. As a result, the WhyML code has two variants of each function, e.g., `add_n` and `add_n_in_place`. The non-in-place variant is simply a wrapper over a generic function that assumes all parameters are aliased. However, the wrappers are inlined at extraction. Therefore, the extracted code only has one version of each function, which can be used in the same ways as its GMP counterpart.

4.2.3 Schoolbook multiplication

I have implemented several algorithms for integer multiplication. For smaller integers (fewer than 30 limbs), the fastest is the schoolbook one, which has complexity $O(n^2)$. For larger operands, Toom-Cook multiplication is used (Sec. 4.4), as it has better asymptotic complexity.

Let us first consider the auxiliary function `addmul_1` (Alg. 3). It multiplies a big integer a by a limb y and adds the result to r, without modifying the

Algorithm 3 Multiply-and-add.

```
Require: $\texttt{valid}(r, m), \texttt{valid}(a, m)$
Ensure: $\texttt{value}(r, m) + \beta^m \cdot \texttt{result} = \texttt{value}(\texttt{old}\ \ r, m) + \texttt{value}(a, m) \cdot y$
Ensure: $\forall j.\ j < 0 \lor m \leq j \implies r[j] = (\texttt{old}\ \ r)[j]$
  function ADDMUL_1$(r, a, m, y)$
    $c, i = 0$
    while $i < m$ do
      $x \leftarrow a[i]$
      $z \leftarrow r[i]$
      $(l, h) \leftarrow$ MUL_DOUBLE$(x, y)$                ▷ $l + \beta \cdot h = x \cdot y$
      $l' \leftarrow$ ADD_MOD$(l, c)$
      $c' \leftarrow$ (if $l' < c$ then $1$ else $0) + h$          ▷ $l' + \beta \cdot c' = x \cdot y + c$
      $l'' \leftarrow$ ADD_MOD$(l', z)$
      $c'' \leftarrow$ (if $l'' < z$ then $1$ else $0) + c'$      ▷ $v + \beta \cdot c'' = z + x \cdot y + c$
      $r[i] \leftarrow v$
      $c \leftarrow c''$
      $i \leftarrow i + 1$
    return $c$
```

contents of r outside the area of the addition. It returns the most significant limb of the sum.

The loop invariants of `addmul_1` are:

1) $0 \leq i \leq m$,

2) $\texttt{value}(r, i) + c\beta^i = \texttt{value}(\texttt{old}\ \ r, i) + \texttt{value}(a, i) \cdot y$,

3) $\forall j.\ j < 0 \lor m \leq j \implies r[j] = (\texttt{old}\ \ r)[j]$.

Let us explain why the computations of c' and c'' do not overflow. As $l + \beta h \leq (\beta - 1)^2$, we have $h < \beta - 1$, so the computation of c' cannot overflow. Furthermore, if $c' = \beta - 1$, then we have $l' + \beta(\beta - 1) = x \cdot y + c \leq \beta(\beta - 1)$, so $l' = 0$. As a result, the computation of c'' does not overflow either.

Algorithm 4 Schoolbook multiplication of two integers.

```
Require: $0 < n \leq m$
Require: $\texttt{valid}(a, m), \texttt{valid}(b, n), \texttt{valid}(r, m + n)$
Ensure: $\texttt{value}(r, m + n) = \texttt{value}(a, m) \cdot \texttt{value}(b, n)$
  function MUL_BASECASE$(r, a, m, b, n)$
    $y \leftarrow b[0]$
    $r[m] \leftarrow$ MUL_1$(r, a, m, y)$       ▷ $\texttt{value}(r, m + 1) = \texttt{value}(a, m) \cdot b[0]$.
    $p \leftarrow r + 1$
    $i \leftarrow 1$
    while $i < n$ do
      $y \leftarrow b[i]$
      $p[m] \leftarrow$ ADDMUL_1$(p, a, m, y)$                   ▷ See Alg. 3.
      $i \leftarrow i + 1$
      $p \leftarrow p + 1$
```

Let us now consider the main function that implements schoolbook multiplication (Alg. 4). It uses the function `addmul_1`, as well as another function called `mul_1`. It is similar to `addmul_1`, but overwrites `r` rather than adding the product to it. At each iteration of the main loop of `mul_basecase`, one limb of the second operand is multiplied by the entire first operand, and the product is added to the result (shifted appropriately).

The loop invariants are as follows:

1) $1 \leq i \leq n$,

2) $\texttt{value}(r, m+i) = \texttt{value}(a, m) \cdot \texttt{value}(b, i)$,

3) $p = r + i$.

It is easy to see that the postcondition follows from the invariants. The fact that the invariants are maintained follows from the specification of the auxiliary function `addmul_1`.

Indeed, if we pose r' the state of r at the beginning of a loop iteration, we have at the end of the loop iteration:

$$\begin{aligned}
&\texttt{value}(r, m+i+1) \\
&= \texttt{value}(r, i) + \beta^i \cdot \texttt{value}(r+i, m+1) && \text{decomposition} \\
&= \texttt{value}(r', i) + \beta^i \cdot \texttt{value}(r+i, m+1) && \text{no writes in } r(0,i) \\
&= \texttt{value}(r', i) + \beta^i \cdot (\texttt{value}(r'+i, m) + \texttt{value}(a, m) \cdot y) && \text{postcondition of } \texttt{addmul\_1} \\
&= \texttt{value}(r', m+i) + \beta^i \cdot \texttt{value}(a, m) \cdot y && \text{recomposition} \\
&= \texttt{value}(a, m) \cdot \texttt{value}(b, i) + \beta^i \cdot \texttt{value}(a, m) \cdot y && \text{loop invariant} \\
&= \texttt{value}(a, m) \cdot (\texttt{value}(b, i) + \beta^i \cdot y) \\
&= \texttt{value}(a, m) \cdot \texttt{value}(b, i+1). && \text{recomposition}
\end{aligned}$$

4.2.4 Division

Long division consists in computing the quotient and remainder of the division of big integers of arbitrary sizes. It is a significantly more complex problem than long addition and multiplication. While the GMP algorithm that we verified is a variation on the schoolbook algorithm, it is thoroughly optimized to the point of making it hard to understand and prove. The general-case long division function is about 50-line long,[1] and the code extracted from my implementation is about 80-line long. However, my Why3 proof for it is about 2000-line long.

Let us first review a more naïve algorithm: Knuth's Algorithm D [57, p. 257] (see also [94]), shown in Alg. 5. I did not use this algorithm in my development, but it is simple enough to explain the core ideas more easily.

We assume a primitive `div_2by1` that divides a 2-limb integer by a 1-limb integer and returns the quotient. It has no WhyML code and we assume that the hardware provides such a function. It is the only division primitive used by the functions in this section.

```
val div_2by1 (l h d:limb) : limb
  requires { to_int h < to_int d }
  ensures { to_int result =
    div (to_int l + radix * to_int h) (to_int d) }
```

[1] `mpn/generic/sbpi1_div_qr.c` in GMP 6.1.2

Algorithm 5 Knuth's Algorithm D.

```
Require: $m \geq n > 0, \texttt{valid}(a, m), \texttt{valid}(d, n), \texttt{valid}(q, m - n), \texttt{valid}(r, n)$
Require: $d[n - 1] \geq \beta/2$
Require: $\texttt{value}(a + m - n, n) < \texttt{value}(d, n)$    ▷ Otherwise an extra quotient limb is needed.
Ensure: $\texttt{value}(a, m) = \texttt{value}(d, n) \cdot \texttt{value}(q, m - n) + \texttt{value}(r, n)$
Ensure: $\texttt{value}(r, n) < \texttt{value}(d, n)$
function AlgorithmD$(q, r, a, d, m, n)$
    for $j = m - n - 1$ downto $0$ do
        $\hat{q} \leftarrow$ DIV_2BY1$(a[j + n - 1], a[j + n], d[n - 1])$ ▷ Candidate quotient limb.
        $\hat{r} \leftarrow \beta a[j + n] + a[j + n - 1] - d[n - 1] \cdot \hat{q}$    ▷ Candidate remainder.
adjust:
        if $\hat{q} \cdot d[n - 2] > \beta \cdot \hat{r} + a[j + n - 2]$ then
            $\hat{q} \leftarrow \hat{q} - 1$                    ▷ Quotient is too large; adjust.
            $\hat{r} \leftarrow \hat{r} + d[n - 1]$
            if $\hat{r} < \beta$ then goto adjust                ▷ Happens at most once.
        $b \leftarrow$ SUBMUL_1$(a + j, a + j, d, \hat{q}, n)$        ▷ Subtract $\overline{d} \cdot \hat{q}$ from $a$.
        $q[j] \leftarrow \hat{q}$
        if $b > 0$ then      ▷ There was a borrow, the quotient was too large.
            $q[j] \leftarrow q[j] - 1$
            $c \leftarrow$ ADD_N$(a + j, a + j, d, n)$
            $a[j + n] \leftarrow a[j + n] + c$                  ▷ Propagate the carry.
    for $i = 0$ to $n - 1$ do      ▷ The remainder is written in $a$, copy it to $r$.
        $r[i] \leftarrow a[i]$
    return
```

The algorithm consists in computing the limbs of the quotient one by one, starting with the most significant. The numerator is overwritten at each step to contain the partial remainder.

At each iteration of the loop, we compute a quotient limb and subtract from the current remainder the product of that quotient limb and the denominator, left-shifted appropriately to cancel out the most significant limb of the current remainder. The function `submul_1` is similar to the function `addmul_1` from the previous section, but subtracts the product instead of adding it.

To compute a quotient limb, a candidate value is first guessed by dividing the two most significant limbs from the current remainder by the most significant limb of the denominator.

This candidate value is then adjusted to match the correct value of the quotient (lines 5-8 and 11-14). This process is called "adjustment step" throughout this section. The algorithms that are actually implemented in GMP are variants of Algorithm D that try to minimize the number of adjustments that occur.

This is where the requirement that $d[n-1] \geq \beta/2$ comes into play. When this is the case (we call such a denominator *normalized*) then the initial 2-by-1 division gives a good approximation of the target quotient limb.

Definition 1. *An integer* $\overline{p[0]\dots p[n-1]}$ *is said to be* normalized *when* $p[n-1] \geq \beta/2$.

```
predicate normalized (x:ptr limb) (sz:int32) =
  valid x sz ∧ x[x.offset + sz - 1] ≥ div radix 2
```

More precisely, as shown by Knuth [57, p. 257, Theorem B], the candidate quotient is at most too large by 2, under the condition that the denominator is normalized. The denominator being normalized is therefore a precondition of Knuth's algorithm, and of the other division algorithms in this section for similar reasons.

In the general case, we remark that an integer is normalized if and only if its most significant bit is set to 1. The denominator can therefore be normalized by counting the leading zeros in the denominator, shifting the numerator and denominator by that amount (the denominator is then normalized), calling a division procedure, and correcting the output by shifting the remainder to the right by the same amount.

This normalization is done by a wrapper around the main division primitive. This wrapper is the function that is exposed to the user. The version of the wrapper that I verified is very simple and only performs this normalization, so it is not worth discussing here. GMP's version also implements an alternative algorithm that is not needed for correctness, but that is faster than the default one when the divisor is close in length to the dividend. I have not implemented and verified this alternate algorithm yet; the more general algorithm is used in all cases in WhyMP. In the rest of this section, we will continue to assume the divisor normalized.

General case algorithm

GMP does not use Knuth's algorithm, but a similar one that uses a 3-by-2 division to compute each quotient limb (Alg. 6). Let us now discuss the differences between this algorithm and Algorithm D.

Algorithm 6 General case long division.

```
Require: $m \geq n \geq 3, \texttt{valid}(a, m), \texttt{valid}(d, n), \texttt{valid}(q, m-n)$
Require: $d[n-1] \geq \beta/2$
Require: $\texttt{value}(a+m-n, n) < \texttt{value}(d, n)$    ▷ Otherwise an extra quotient limb is needed
Ensure: $\texttt{value}(\texttt{old } a, m) = \texttt{value}(q, m-n) \cdot \texttt{value}(d, n) + \texttt{value}(a, n)$
Ensure: $\texttt{value}(a, n) < \texttt{value}(d, n)$
function DIV_SB_QR$(q, a, d, m, n)$
    $v \leftarrow$ RECIPROCAL_WORD_3BY2$(d[n-1], d[n-2])$
    $x \leftarrow a[m-1]$
    $i = m - n$
    while $i > 0$ do
        $i \leftarrow i - 1$
        if $x = d[n-1]$ and $a[n+i-1] = d[n-2]$ then    ▷ Unlikely.
            $\hat{q} \leftarrow \beta - 1$
            SUBMUL_1$(a+i, d, n, \hat{q})$    ▷ We know the result is $d[n-1]$.
            $x \leftarrow a[n+i-1]$
        else
            $(\hat{q}, l, x) \leftarrow$ DIV3BY2_INV$(x, a[n+i-1], a[n+i-2], d[n-1], d[n-2], v)$
            $b \leftarrow$ SUBMUL_1$(a+i, d, n-2, \hat{q})$
            $b_1 \leftarrow (l < b)$    ▷ Last two steps of the subtraction are inlined.
            $a[i+n-2] \leftarrow l - b \mod \beta$
            $b_2 \leftarrow (x < b_1)$
            $x \leftarrow x - b_1 \mod \beta$    ▷ Finish subtraction.
            if $b_2 \neq 0$ then    ▷ Unlikely, and $b_2 = 1$.
                $\hat{q} \leftarrow \hat{q} - 1$    ▷ We only need to adjust by 1.
                $c \leftarrow$ ADD$(a+i, a+i, d, n-1)$    ▷ Add only over $n-1$ limbs.
                $x \leftarrow x + d[n-1] + c \mod \beta$    ▷ The carry out is always 1.
        $q[i] \leftarrow \hat{q}$
    $a[n-1] \leftarrow x$
```

First, there is an extra local variable x. It is really a proxy for the most significant limb in the current remainder, in the sense that whenever we would read from $a[n+i]$, we take x instead. Thus, instead of being stored in $\overline{a[i]\dots a[n+i]}$, the low n limbs of the current remainder are stored in $\overline{a[i]\dots a[n+i-1]}$ and the most significant limb is stored separately in the variable x. This saves a few memory accesses, and it can be done because the most significant limb of the current remainder is no longer needed after the current loop iteration.

We are now better equipped to express the main loop invariants.

Let A be equal to the initial value of $\texttt{value}(a, m)$. The following invariants are maintained in the main loop:

1) $A = \texttt{value}(d, n) \cdot \beta^i \cdot \texttt{value}(q+i, m-n-i) + \texttt{value}(a, n+i-1) + \beta^{n+i-1} \cdot x$,

2) $\texttt{value}(a+i, n-1) + \beta^{n-1} \cdot x < \texttt{value}(d, n)$,

3) $d[n-2] + \beta d[n-1] \geq a[n+i-2] + \beta x$ (implied by the previous two).

An important difference with Algorithm D is that instead of dividing the two most significant limbs of the numerator by the most significant limb of the denominator, we divide the three most significant limbs of the former by the two most significant limbs of the latter. This means that the adjustment step is much more efficient than that of Algorithm D. Indeed, the candidate quotient obtained this way is very likely to be correct (Lemma 4).

Lemma 4. *Let $\hat{q}$ and $\overline{r[0]r[1]}$ the quotient and remainder of the division of $\overline{a[n+i-2]a[n+i-1]x}$ by $\overline{d[n-2]d[n-1]}$.*
If $\hat{q} \cdot \overline{d[0]\dots d[n-1]} > \overline{a[i]\dots a[n+i-1]x}$, then $r[1] = 0$.

Proof. We have $\overline{a[n+i-2]a[n+i-1]x} = \overline{d[n-2]d[n-1]}\hat{q} + \overline{r[0]r[1]}$.
We also have $\overline{a[i]\dots a[n+i-1]x} \geq \beta^{n-2}\overline{a[n+i-2]a[n+i-1]x}$.
If $\hat{q} \cdot \overline{d[0]\dots d[n-1]} > \overline{a[i]\dots a[n+i-1]x}$, this implies:

$$\hat{q} \cdot \overline{d[0]\dots d[n-1]} > \beta^{n-2}(\overline{d[n-2]d[n-1]}\hat{q} + \overline{r[0]r[1]})$$

However,

$$\begin{aligned}\hat{q} \cdot \overline{d[0]\dots d[n-1]} &= \hat{q} \cdot \overline{d[0]\dots d[n-3]} + \beta^{n-2}\overline{d[n-2]d[n-1]}\hat{q} \\ &< \beta^{n-1} + \beta^{n-2}\overline{d[n-2]d[n-1]}\hat{q}.\end{aligned}$$

Therefore we must have $\beta^{n-2}\overline{r[0]r[1]} < \beta^{n-1}$, which implies $r[1] = 0$. □

This means that the borrow at line 18 is only non-zero when the high limb of the remainder returned by the 3-by-2 division at line 12 is zero, which is intuitively rare (if the outcomes were evenly distributed, the probability would be $1/\beta$). This lemma does not have an equivalent in the Why3 proof, as it is not needed to prove functional correctness. However, the fact that the `if` statement at line 18 is very likely to take the empty "else" branch is signaled in GMP's original source code, which reads: `if (UNLIKELY (cy != 0))`. The `UNLIKELY` macro is unfolded to the compiler built-in `__builtin_expect`, which instructs the compiler to favour the likelier branch in the assembly code. The built-in is also present in the extracted code of WhyMP.

Not only is the initial guess for the quotient very likely correct, but when it is not, it is only too large by 1 and we can correct it with a single incrementation.

Compare this to Algorithm D, where two separate blocks were dedicated to adjusting the quotient, and one of which could be executed twice. The following lemma justifies that the candidate quotient is at worst too large by 1, which justifies that the adjustment step at line 18 is needed at most once.

Lemma 5. *Let $\hat{q}$ and $\overline{r[0]r[1]}$ the quotient and remainder of the division of $\overline{a[n+i-2]a[n+i-1]x}$ by $\overline{d[n-2]d[n-1]}$.*
Then $(\hat{q}-1)\cdot\overline{d[0]\dots d[n-1]} \leq \overline{a[i]\dots a[n+i-1]x}$.

Proof. We have $\hat{q}\cdot\overline{d[n-2]d[n-1]} \leq \overline{a[n+i-2]a[n+i-1]x}$, so:

$$\begin{aligned}
&(\hat{q}-1)\overline{d[0]\dots d[n-1]}\\
&\leq (\hat{q}-1)\overline{d[0]\dots d[n-3]} + \beta^{n-2}(\hat{q}-1)\overline{d[n-2]d[n-1]}\\
&< \beta^{n-1} + \beta^{n-2}\overline{a[n+i-2]a[n+i-1]x} - \beta^{n-2}\overline{d[n-2]d[n-1]}\\
&< \beta^{n-2}\overline{a[n+i-2]a[n+i-1]x} \quad \text{(we have } d[n-1] \geq \beta/2\text{, and assume } \beta > 2\text{)}\\
&\leq \overline{a[i]\dots a[n+i-1]x}. \quad \square
\end{aligned}$$

□

The remainder of the 3-by-2 division is also used: instead of a simple long subtraction over a length n like in Algorithm D, we perform a long subtraction over a length $n-2$ only and inline the last two steps. These last two steps consist simply in propagating the borrow from the previous subtraction, as the result of the 3 most significant limbs of subtraction are known to be $\overline{lx0}$ in the absence of borrow (the postcondition of the division is exactly that $\overline{a[n+i-2]a[n+i-1]x} = \hat{q}\cdot\overline{d[n-2]d[n-1]} + \overline{lx}$). We then propagate the borrow on $\overline{lx0}$. Hence, lines 13 to 17 are equivalent to computing the subtraction

$$\overline{a[i]\dots a[n+i-1]x} - \hat{q}\cdot\overline{d[0]\dots d[n-1]}$$

returning b_2 as borrow and writing the result in $\overline{a[i]\dots a[n+i-2]x}$ (one limb fewer).

If $b_2 = 0$, the first invariant is maintained. Otherwise, there is an adjustment to make (line 18): if the subtraction overflows, our candidate quotient $\hat{q}$ was too large, we subtract 1 from it and add back $\texttt{value}(d,n)$ to the remainder. This addition is done at lines 20-21. The last limb is added separately because we save a few operations by ignoring the carry out (we know it is equal to 1).

It is efficient to do the subtraction first and potentially backtrack on it later because it happens very rarely (Lemma 4), and evaluating the subtraction makes it easy to check if the candidate quotient was correct with a simple integer comparison.

The specification of the 3-by-2 division primitive ensures that $\overline{lx}$ is less than $\overline{d[n-2]d[n-1]}$, hence the second and third invariants are maintained if there is no adjustment. If there is an adjustment, the addition at line 20 may overflow, but the value of the top two limbs of the remainder will still be $\overline{lx}+1 \leq \overline{d[n-2]d[n-1]}$. This shows that the third invariant still holds in this case. For the second invariant, we use the fact that $\texttt{value}(a+i,n-1)+\beta^{n-1}x-\beta^{n}b_2$ is negative if $b_2 \neq 0$, so adding back $\texttt{value}(d,n)$ maintains the second invariant.

There is an extra contingency in the main loop: when the two most significant limbs of the denominator and the current numerator are identical, then we can skip the 3-by-2 division and adjustment step. Indeed, the candidate quotient output by the division would necessarily be β, but the third invariant ensures that the long subtraction that would follow would have a non-zero borrow, and that the adjustment step would knock the candidate quotient down to $\beta-1$. Therefore, we immediately write in $\beta-1$ as quotient limb. With this case out of the way, we may write the 3-by-2 division in such a way that its quotient output is always a single limb, which would otherwise not be true in general.

3-by-2 division

Let us now take a closer look at the 3-by-2 division subroutine used by the division algorithm. It was introduced by Möller and Granlund [73].

```
predicate reciprocal_3by2 (v dh dl:limb) =
  v = div (radix*radix*radix -1) (dl + radix * dh) - radix

let div3by2_inv (uh um ul dh dl v: limb) : (limb,limb,limb)
  requires { dh ≥ div radix 2 }
  requires { reciprocal_3by2 v dh dl }
  requires { um + radix * uh < dl + radix * dh }
  returns { q, rl, rh → q * (dl + radix * dh) + rl + radix * rh
                       = ul + radix * (um + radix * uh) }
  returns { q, rl, rh → 0 ≤ rl + radix * rh < dl + radix * dh }
```

The algorithm takes a precomputed pseudo-inverse v of the denominator $\overline{d_l d_h}$ as an extra parameter. More precisely,

$$v = \left\lfloor \frac{\beta^3 - 1}{d_l + \beta d_h} \right\rfloor - \beta.$$

The reason it is precomputed and passed as a parameter rather than computed on the fly is that the caller function (Alg. 6) always uses the same denominator over a long division, so it is much more efficient to compute the pseudo-inverse only once.

The algorithm essentially consists in multiplying $\overline{d_l d_h}$ by its pseudo-inverse v and then performing some simple adjustments. Remarkably, this means that no division primitive is used. As division primitives tend to be much more expensive than additions or multiplications, this makes the algorithm a very efficient way to perform a 3-by-2 division.

The "trick" is that the computation of the pseudo-inverse itself does use a division primitive. In fact, computing the pseudo-inverse is about as complex as the 3-by-2 division proper because of this division. However, over an m-by-n long division, $m-n$ short divisions are performed, all with the same precomputed pseudo-inverse, which amortizes that cost.

While the algorithm itself is short, its proof is non-trivial. The hardest part was directly taken from Möller and Granlund's on-paper proof [73, Theorem 3] and transcribed into a hundred-line Why3 assertion. The algorithm that computes the pseudo-inverse with a single 2-by-1 division primitive was also taken from their paper.

Smaller cases: $n=1$ and $n=2$

The general case algorithm only handles the case where the denominator has length 3 or more. Different algorithms are used for smaller denominators. When the divisor is exactly one limb long, the schoolbook algorithm is used (Alg. 7).

Algorithm 7 Schoolbook division by a 1-limb number.

```
Require: $m \geq 1, \texttt{valid}(q,m), \texttt{valid}(a,m), d \geq \beta/2$
Ensure: $\texttt{value}(a,m) = \texttt{result} + d \cdot \texttt{value}(q,m)$
Ensure: $\texttt{result} < d$
function DIVREM_1$(q,a,m,d)$
    $v \leftarrow$ INVERT_LIMB$(d)$
    $r \leftarrow 0$
    $i \leftarrow m-1$
    while $i \geq 0$ do
        $(\hat{q},\hat{r}) \leftarrow$ DIV2BY1_INV$(r, a[i], d, v)$          ▷ Divide $\overline{a[i]r}$ by $d$
        $q[i] \leftarrow q$
        $r \leftarrow \hat{r}$
        $i \leftarrow i-1$
    return $r$
```

The variable r is the partial remainder, in the sense of the following loop invariants:

1) $\texttt{value}(a+i+1, m-i-1) = d \cdot \texttt{value}(q+i+1, m-i-1) + r$,

2) $r < \beta$.

Let us prove that the first invariant is maintained. Assume $\texttt{value}(a+i+1, m-i-1) = d \cdot \texttt{value}(q+i+1, m-i-1) + r$ and $r < \beta$. Let $\hat{q}, \hat{r}$ such that $\hat{q}d + \hat{r} = a[i] + \beta r$. Then after $q[i] \leftarrow q$,

$$\begin{aligned}
&\texttt{value}(a+(i-1)+1, m-(i-1)-1) \\
&= \texttt{value}(a+i, m-i) \\
&= a[i] + \beta\texttt{value}(a+i+1, m-i-1) && \texttt{[value\_sub\_head]} \\
&= a[i] + \beta d \cdot \texttt{value}(q+i+1, m-i-1) + \beta r \\
&= \hat{q}d + \hat{r} + \beta d \cdot \texttt{value}(q+i+1, m-i-1) \\
&= d \cdot \texttt{value}(q+i, m-i) + \hat{r} \\
&= d \cdot \texttt{value}(q+(i-1)+1, m-(i-1)-1) + \hat{r}.
\end{aligned}$$

Finally, similarly to the general algorithm, we precompute a pseudo-inverse v of d for the 2-by-1 division, and subsequently use a 2-by-1 division algorithm that uses no division primitive. This time, we have $v = \lfloor(\beta^2-1)/d\rfloor - \beta$.

```
predicate reciprocal (v d:limb) =
v = (div (radix*radix - 1) d) - radix

let div2by1_inv (uh ul d v:limb) : (limb,limb)
  requires { d ≥ div radix 2 }
  requires { uh < d }
  requires { reciprocal v d }
  returns { q, r →  q * d + r = ul + radix * uh }
  returns { q, r → 0 ≤ r < d }
```

When $n = 2$, a very similar algorithm is used (Alg. 8). The only difference is that 3-by-2 division is used at each loop iteration. Similarly to the $n = 1$ case, $\overline{lh}$ is the current remainder, and the loop invariants are as follows:

Algorithm 8 Schoolbook division by a 2-limb number.

```
Require: $m \geq 2, \texttt{valid}(q, m-2), \texttt{valid}(a, m), \texttt{valid}(d, 2), d[1] \geq \beta/2$
Require: $\texttt{value}(a+m-2, 2) < \texttt{value}(d, 2)$    ▷ Otherwise an extra quotient
    limb is needed.
Ensure: $\texttt{value}(\texttt{old}\ a, m) = \texttt{value}(d, 2) \cdot \texttt{value}(q, m-2) + \texttt{value}(a, 2)$
Ensure: $\texttt{value}(a, 2) < \texttt{value}(d, 2)$
function DIVREM_2$(q, a, m, d)$
    $v \leftarrow$ RECIPROCAL_3BY2$(d[1], d[0])$
    $h \leftarrow a[m-1]$
    $l \leftarrow a[m-2]$
    $i \leftarrow m-2$
    while $i > 0$ do
        $(\hat{q}, l, h) \leftarrow$ DIV3BY2_INV$(h, l, a[i-1], d[1], d[0], v)$
        $i \leftarrow i-1$
        $q[i] \leftarrow \hat{q}$
    $a[1] \leftarrow h$
    $a[0] \leftarrow l$
```

1) $\texttt{value}(a, m) = \texttt{value}(a, i) + \beta^i(\texttt{value}(q+i, m-i-2) \cdot \texttt{value}(d, 2) + l + \beta h)$,

2) $l + \beta h < \texttt{value}(d, 2)$.

Let us now discuss the differences between these two special cases on the one hand, and the general case algorithm on the other hand. The adjustment step that is found in Algorithm D and other general long division algorithms is not needed in any of these two special cases. Indeed, what makes the adjustment necessary in Algorithm D (and in GMP's algorithm) is that the candidate quotient is computed using an incomplete denominator. When the denominator has length 1 or 2, the division that occurs at each loop iteration uses the full denominator. Even though the numerator still has arbitrary length, this does not cause errors on the candidate quotient.

Another difference is that no subtraction is needed to compute the partial remainders in the two short cases. The corresponding operation in Alg. 7 is simply the $r \leftarrow \hat{r}$ assignment at line 8. Indeed, we have $d\hat{q} + \hat{r} = \overline{a[i]r}$ from the 2-by-1 division, and the long subtraction that we would perform with Algorithm D would be exactly $\overline{a[i]r} - d\hat{q} = \hat{r}$, so decrementing i and assigning r to $\hat{r}$ does the same thing and saves a subtraction. Again, this is due to the fact that we use the whole denominator at each loop iteration.

4.3 Modular exponentiation

Consider the following problem. Given three positive large integers b, e and m, how to compute b^e modulo m? In the non-modular case, without considering the multiple-precision representation of numbers, the usual basic algorithm for fast exponentiation is the binary square-and-multiply one. We consider the binary decomposition of e, starting with the most significant bit. We accumulate the result in some variable r, initialized at $r = e^0 = 1$. For each bit of e, we square r, and then multiply r by e if the current bit is 1. GMP's algorithm is based on this approach, but is optimized in many ways. It assumes that m is odd, for reasons that will be made clearer in the next paragraph. A separate algorithm is provided for the case where m is even. Note that in cryptographic applications, m is odd in the vast majority of cases (computations are typically done modulo an odd prime number).

In Sec. 4.3.1, we first present various primitives that perform computations in Montgomery form, which is used in GMP's exponentiation algorithm. We then go over an auxiliary function that computes the inverse of a limb modulo β (Sec. 4.3.2), and auxiliary functions that perform bit-level computations (Sec. 4.3.3). Finally, we move on to the main algorithm (Sec. 4.3.4) and discuss its side-channel resistant variant (Sec. 4.3.5).

4.3.1 Computations in Montgomery form

The basic square-and-multiply algorithm computes a simple exponentiation. However, what we are trying to compute is not b^e, but its residue modulo m. Of course, simply computing b^e and then reducing it modulo m is not an acceptable algorithm, as b^e would not fit in memory for even moderately large values of e. A simple approach consists in applying the basic algorithm while reducing r modulo n at regular intervals, such as before each square operation. This algorithm works, but a reduction modulo m is essentially a division, which is more costly than a multiplication of two numbers of the same length as m. Therefore, it would be better to find a way to minimize the numbers of divisions. GMP's algorithm achieves this using Montgomery's method [74]. Fix R coprime with m. We call the *Montgomery form* of an integer a the representative of aR modulo m. Since R is coprime with m, multiplication by R is an isomorphism on the additive group $\mathbb{Z}/n\mathbb{Z}$, that is, addition and subtraction in Montgomery form are the same addition and subtraction. Multiplication is more complicated. Indeed, if we simply multiply aR by bR modulo m, the product is congruent to abR^2, rather than abR. However, Montgomery's REDC algorithm computes a division by R modulo m using only divisions by R, rather than m. Therefore, it is very fast assuming a well-chosen R, such as a power of β. This is the main reason why the algorithm requires m to be odd, which makes it coprime with β^n for all positive n. Assuming a fast REDC algorithm, computing a chain of multiplications modulo m thus becomes affordable. Let us now go over Montgomery's REDC algorithm (Alg. 9), taken directly from his article with some notation changes [74].

Let us succintly justify that this algorithm is correct. First, notice that $am \equiv xmm' \equiv -x \mod R$. Therefore, R divides $x + am$, so the division at line 3 is exact. Second, $tR \equiv x \mod m$, so $t \equiv xR^{-1} \mod m$. Finally, $0 \leq x < Rm$ and $0 \leq a < R$, so $0 \leq x + am < 2Rm$. Therefore, $0 \leq t < 2m$, so the result lies in $[0, m-1]$.

GMP implements several versions of the REDC algorithm adjusted for multiprecision integers. Due to time constraints, I only verified the one best suited to smaller integers. Its pseudocode is in Alg. 10. Rather than taking a radix R as parameter, it takes an integer n and sets $R = \beta^n$. The specification is otherwise relatively similar to that of Alg. 9. There are a few differences. First, v is only an inverse of `value`(m, n) modulo β, rather than modulo β^n. It is cheaper to compute this way, and we verify that it is precise enough. Second, the specification is a bit weaker. The precondition $u < Rm$ is not necessary,

Algorithm 9 Montgomery's REDC algorithm (infinite precision case).

```
Require: $mm' \equiv -1 \mod R$
Require: $m \wedge R = 1$
Require: $0 \le x < Rm$
Ensure: $0 \le \texttt{result} < m$
Ensure: $\texttt{result} \equiv xR^{-1} \mod m$
function REDC($x$, $R$, $m$, $m'$)
    $a \leftarrow (x \mod R)m' \mod R$
    $t \leftarrow (x + am)/R$                          ▷ The division is exact.
    if $t \ge m$ then
        return $t - m$
    else
        return $t$
```

although it is needed to prove that the result is small. One call to this function in the main exponentiation algorithm does in fact violate this condition. In the first postcondition, both sides of the equality have been multiplied by β^n. This is equivalent, as β^n and $\texttt{value}(m, n)$ are coprime. However, this version of the postcondition is more elementary, as it does not require the user to define division modulo an integer in Why3's logic. Finally, note that the result is not normalized, as the result $\texttt{value}(r, n)$ is not necessarily smaller than $\texttt{value}(m, n)$.

Algorithm 10 Montgomery's REDC algorithm, in GMP.

```
Require: $n > 0$
Require: $\texttt{valid}(m, n) \wedge \texttt{valid}(u, 2n) \wedge \texttt{valid}(r, n)$
Require: $\texttt{value}(m, n)$ odd
Require: $v \cdot \texttt{value}(m, n) \equiv -1 \mod \beta$
Ensure: $\texttt{value}(\texttt{old}\ u, 2n) \equiv \beta^n \texttt{value}(r, n) \mod \texttt{value}(m, n)$
Ensure: $\forall j.\ j < 0 \vee n \le j \implies r[j] = (\texttt{old}\ r)[j]$
Ensure: $\texttt{value}(\texttt{old}\ u, 2n) < \beta^n \texttt{value}(m, n) \implies \texttt{value}(r, n) < 2\texttt{value}(m, n)$
function REDC_1($r, u, m, n, v$)
    for $j = 0$ to $n - 1$ do
        $c \leftarrow$ ADDMUL_1($u + j, m, n, u[j]v \bmod \beta$)     ▷ After this, $u[j] = 0$.
        $u[j] \leftarrow c$
    $c \leftarrow$ ADD_N($r, u, u + n, n$)
    if $c \ne 0$ then                          ▷ $\texttt{value}(r, n)$ is too large, adjust.
        SUB_N($r, r, m, n$)
```

Fix $U = \texttt{value}(u, 2n)$ at the start of the algorithm, and $M = \texttt{value}(m, n)$. Recall how Alg. 9 adds a multiple of m to its input to make it divisible by R without changing its value modulo m. Similarly, the idea of this algorithm is to add multiples of M to U to make it divisible by β, then β^2, and so on until β^n. However, the situation is more complicated, because we want to store the carries of the successive additions without needing extra space. The algorithm reuses the low limbs of u that would otherwise be zeroes, and stores the carries there. In the end, the loop invariants are as follows. The second invariant is needed to show that the final adjustment step is sufficient.

1) $\beta^j \texttt{value}(u+j, 2n-j) + \beta^n \texttt{value}(u, j) \equiv U \mod M$,

2) $U \leq \beta^j \texttt{value}(u+j, 2n-j) + \beta^n \texttt{value}(u, j) < U + \beta^j \cdot M$.

Let us first check that the loop invariants are valid. For both invariants, the initialization $j = 0$ is trivial.

Invariant preservation Assume the invariants are valid for some j with $0 \leq j < n$. Let us prove that they are valid for $j + 1$.

For the sake of notation, let us call u' the state of u at the beginning of the loop iteration. The invariant hypothesis gives us:

$$\beta^j \texttt{value}(u'+j, 2n-j) + \beta^n \texttt{value}(u', j) \equiv U \mod M.$$

Let us first show that the addition at line 3 cancels out $u[j]$. Indeed, the quantity that is being added to $u + j$, modulo β, is $\texttt{value}(m, n) \cdot v \cdot u'[j]$, which is congruent to $-u'[j]$ modulo β by definition of v.

After the addition at line 3, we have $\texttt{value}(u+j, 2n-j) + c \cdot \beta^n \equiv \texttt{value}(u'+j, 2n-j) \mod M$, as the quantity that is added to $u + j$ is a multiple of M. Moreover, $u[j]$ is now 0, so $\texttt{value}(u+j, 2n-j) = \beta \cdot \texttt{value}(u+j+1, 2n-(j+1))$. In the end, we have

$$c \cdot \beta^{n+j} + \beta^{j+1} \cdot \texttt{value}(u+j+1, 2n-(j+1)) \equiv \beta^j \texttt{value}(u'+j, 2n-j) \mod M.$$

More precisely, if a is the product of $u[j] \cdot v$ modulo β $(0 \leq a < \beta)$, we have

$$c \cdot \beta^{n+j} + \beta^{j+1} \cdot \texttt{value}(u+j+1, 2n-(j+1)) = \beta^j(\texttt{value}(u'+j, 2n-j) + aM).$$

After setting $u[j]$ to c at line 4, the value of $(u + j + 1, 2n - (j + 1))$ is of course unchanged, and we have $\beta^n \cdot \texttt{value}(u, j+1) = \beta^n \cdot \texttt{value}(u', j) + c \cdot \beta^{n+j}$.

Adding the last two equalities, we obtain:

$$\begin{aligned} &\beta^{j+1} \cdot \texttt{value}(u+j+1, 2n-(j+1)) + \beta^n \cdot \texttt{value}(u, j+1) \\ &= \beta^j \cdot \texttt{value}(u'+j, 2n-j) + \beta^n \cdot \texttt{value}(u', j) + a \cdot \beta^j \cdot M. \end{aligned}$$

The difference $a \cdot \beta^j \cdot M$ is divisible by M, so the first invariant is maintained. Let us note A the difference $\beta^j \cdot \texttt{value}(u'+j, 2n-j) + \beta^n \cdot \texttt{value}(u', j) - U$. The second invariant hypothesis gives us $0 \leq A < \beta^j M$, and the new difference is $A + a \cdot \beta^j \cdot M$. Since $0 \leq a \leq \beta - 1$, we get $0 \leq A + a \cdot \beta^j \cdot M < \beta^{j+1} M$, so the second invariant is maintained.

Postconditions Now that the loop invariant is proved by induction, let us show that it implies the postconditions. The second postcondition is clearly correct, as there is never any write in r except in the interval (r, n).

Immediately after the loop, the first invariant yields the following identity:

$$\beta^n(\texttt{value}(u+n, n) + \texttt{value}(u, n)) \equiv U \mod M.$$

So, after the addition at line 5, we have $\beta^n \cdot (\texttt{value}(r, n) + c \cdot \beta^n) \equiv U \mod M$.

Furthermore, the second invariant gives us $\beta^n \cdot (\texttt{value}(r, n) + c \cdot \beta^n) < U + M\beta^n$. Since $U < \beta^{2n}$, we get $\texttt{value}(r, n) + c \cdot \beta^n < \beta^n + M$.

If $c = 0$, the first postcondition is clearly valid. If not, necessarily $c = 1$, so $\texttt{value}(r, n) < M$. Therefore, the subtraction at line 7 offsets c, and the first postcondition is also valid.

Assume now that $U < M\beta^n$. This implies that $\texttt{value}(r, n) + c \cdot \beta^n < 2M$ after the addition at line 5, so the third postcondition is valid regardless of whether c is 0 or 1.

Using the REDC algorithm, we can easily compute multiplications modulo M on numbers in Montgomery form. We simply multiply them (without reducing modulo M) and apply REDC on the result. One remaining issue is how to put numbers in Montgomery form in the first place, that is, from a number x, how to efficiently compute $x\beta^n \mod M$. GMP simply does this the naive way with the self-explanatory `redcify` function, represented in Alg. 11. The division function `div_qr` is a wrapper around the division functions from Sec. 4.2.4, as well as other division functions suited to larger integers that were not implemented in WhyMP due to time constraints.

Algorithm 11 The `redcify` function.

```
Require: $\texttt{valid}(r, n) \wedge \texttt{valid}(u, s) \wedge \texttt{valid}(m, n)$
Require: $n > 0 \wedge s > 0$
Require: $s + n < 2^{32}$    ▷ So that the computation in line 2 does not overflow.
Require: $m[n-1] > 0$
Ensure: $\texttt{value}(r, n) \equiv \beta^n \cdot \texttt{value}(u, s) \mod \texttt{value}(m, n)$
Ensure: $0 \leq \texttt{value}(r, n) < \texttt{value}(m, n)$
1: function REDCIFY$(r, u, s, m, n)$
2:     $t \leftarrow$ TMP_ALLOC$(s + n)$
3:     $q \leftarrow$ TMP_ALLOC$(s + 1)$
4:     ZERO$(t, n)$
5:     COPYI$(t + n, u, s)$    ▷ $\texttt{value}(t, n + s) = \beta^n \cdot \texttt{value}(u, s)$
6:     DIV_QR$(q, r, t, n + s, m, n)$
```

4.3.2 Auxiliary function: limb inversion modulo β

In order to divide a number by β^n modulo some odd integer $M = \texttt{value}(m, n)$, GMP's REDC implementation (Alg. 10) needs an inverse of M modulo β. The computation of this inverse is not trivial. We start by observing that $M \equiv m[0] \mod \beta$, so computing an inverse of $m[0]$ modulo β is enough. A schoolbook method could implement the extended Euclidean algorithm. However, GMP implements a faster algorithm based on Newton's method.

We start with an inverse x_0 of $m[0]$ modulo 2^8, taken from a precomputed table. We successively upgrade this inverse into an inverse x_1 of $m[0]$ modulo 2^{16}, then x_2 modulo 2^{32}, and finally x_3 modulo 2^{64}. This is justified by the following lemma. The formula for x_{i+1} is expressed modulo 2^{64} to take into account potential overflows during the computation.

Lemma 6. *Let p, m, x_i three non-negative integers such that $p \leq 32$ and $mx_i \equiv 1 \mod 2^p$. Let $x_{i+1} \equiv 2x_i - mx_i^2 \mod 2^{64}$. Then $mx_{i+1} \equiv 1 \mod 2^{2p}$.*

Proof. Let d such that $mx_i = 2^p d + 1$. Then $mx_{i+1} \equiv 2mx_i - (mx_i)^2 \equiv 2^{p+1}d + 2 - (2^{2p}d^2 + 2^{p+1}d + 1) \equiv -2^{2p}d^2 + 1 \mod 2^{64}$. As 2^{2p} divides 2^{64}, the equality is also true modulo 2^{2p}. Finally, we find $mx_{i+1} \equiv -2^{2p}d^2 + 1 \equiv 1 \mod 2^{2p}$. □

Note that the formula $x_{i+1} = 2x_i - mx_i^2$ is what you would find when attempting to compute a root of $f(x) = m - \frac{1}{x}$ using Hensel's method. It converges quadratically in general, so this result is not too surprising. Also note that x_{i+1} can be computed from x_i and m using only multiplications and subtractions, rather than the more expensive division. So, we can expect that an algorithm based on this lemma would be faster than simply using Euclid's algorithm, which converges linearly.

A transcription of GMP's modular inverse algorithm can be found in Alg. 12. It assumes that t is a precomputed table of length 128 such that for all x in $[0, 127]$, $(2x+1)t[x] \equiv 1 \mod 2^8$. It is included in the original code as a global variable. The algorithm itself is straightforward. We get x_0 from the global table, and then successively compute x_1, x_2, and x_3 using the formula from the lemma.

Algorithm 12 Inverse of a limb modulo β.

```
Require: $m$ odd
Require: $\beta = 2^{64}$
Ensure: result $\cdot\, m \equiv 1 \mod \beta$
function BINVERT_LIMB($n$)
    $h \leftarrow n/2 \mod 128$                                    ▷ $2h + 1 \equiv n \mod 256$
    $x_0 \leftarrow t[h]$                                           ▷ $x_0 m \equiv 1 \mod 2^8$
    $x_1 \leftarrow$ SUB_MOD(MUL_MOD($2, x_0$), MUL_MOD($x_0$, MUL_MOD($x_0, n$)))
    $x_2 \leftarrow$ SUB_MOD(MUL_MOD($2, x_1$), MUL_MOD($x_1$, MUL_MOD($x_1, n$)))
    $x_3 \leftarrow$ SUB_MOD(MUL_MOD($2, x_2$), MUL_MOD($x_2$, MUL_MOD($x_2, n$)))
    return $x_3$
```

4.3.3 Auxiliary functions: bit-level computations

The modular exponentiation algorithm does not handle the exponent limb by limb, but at the bit granularity. The usual `value` formalism is ill-suited to this. Provided a pointer a is valid over the length $\lceil \frac{b}{64} \rceil$, we represent a as the string of bits $a_0 \ldots a_{b-1}$, with a_0 being the least significant bit. We denote $\underline{a_i \ldots a_j} = \sum_{k=i}^{j} 2^{k-i} a_k$ and we define $\texttt{bitvalue}(a, b) = \underline{a_0 \ldots a_{b-1}} = \sum_{k=0}^{b-1} a_k 2^k$.

Note that `bitvalue` can also be computed in terms of `value`: if $b = 64q + r$, with $0 \leq r < 64$, then $\texttt{bitvalue}(a, b) = \texttt{value}(a, q) + \beta^q (a[q] \mod 2^r)$. In fact, that is the definition of `bitvalue` in the Why3 development.

Let us list a few simple lemmas about `bitvalue`.

Lemma 7. *Let p a pointer valid over a sufficiently large length.*

$$\forall k \geq 0.\ 0 \leq \underline{a_0 \ldots a_{k-1}} < 2^k$$

$$\forall i, j, k.\ 0 \leq i < k \leq j \implies \underline{a_i \ldots a_j} = \underline{a_i \ldots a_{k-1}} + 2^{k-i} \underline{a_k \ldots a_j}$$

$$\forall n, k.\ 64n \geq k \implies \textit{bitvalue}(p, k) = \textit{value}(p, n) \mod 2^k.$$

GMP uses a pair of helper functions to extract a range of bits from a multiprecision integer. The `getbit` function (Alg. 13) returns the $i-1$-th bit of p.

Algorithm 13 Extracting a bit from a large integer.

```
Require: valid$(p, (i + 63)/64)$
Require: $1 \leq i$
Ensure: $0 \leq$ result $< 1$
Ensure: bitvalue$(a, i) =$ bitvalue$(a, i - 1) + 2^{i-1} \cdot$ result
1: function GETBIT($p$, $i$)
2:     $k \leftarrow (i - 1)/64$
3:     $m \leftarrow (i - 1) \mod 64$
4:     return $(p[k] \gg m) \mod 2$
```

The first precondition is equivalent to `valid`$(p, \lceil i/64 \rceil)$, but does not require introducing the concept of rounded up division. The second postcondition is equivalent to saying the result is p_{i-1}, but stating it in this form is more convenient later on. The $\gg$ operator at line 4 denotes a bitwise right shift. The algorithm itself is straightforward. We divide $i - 1$ by 64 to know in which limb the desired bit is, and then we extract the bit from the limb. The right shift puts the desired bit in the least-significant position, and the modulo operator removes all the other bits.

A slightly more complex function is used to extract a range of bits, rather than just one. The `getbits` function (Alg. 14) takes a pointer p, an offset i and a length n, and returns $\underline{p_{i-n} \dots p_{i-1}}$.

Algorithm 14 Extracting a range of bits from a large integer.

```
Require: $1 \leq n < 64$
Require: $0 \leq i$
Require: valid$(p, (i + 63)/64)$
Ensure: $0 \leq$ result $< 2^n$
Ensure: $n \leq i \implies$ bitvalue$(p, i) =$ bitvalue$(p, i - n) + 2^{i-n} \cdot$ result
Ensure: $i < n \implies$ bitvalue$(p, i) =$ result
1: function GETBITS($p$,$i$,$n$)
2:     if $i < n$ then
3:         return $p[0] \mod 2^i$
4:     else
5:         $b \leftarrow i - n$
6:         $k \leftarrow b/64$
7:         $b \leftarrow b \mod 64$
8:         $r \leftarrow p[k] \gg b$
9:         $s \leftarrow 64 - b$                        ▷ $s$ is the number of bits in $r$.
10:        if $s < n$ then                             ▷ We did not get enough bits.
11:            $r \leftarrow r + p[k + 1] \ll s$           ▷ We now have 64 bits.
12:        return $r \mod 2^n$
```

There are three main cases to consider. First, we can have $i < n$. In this case, we simply get the first i bits of $p[0]$. In the second case, the range $p_{i-n} \dots p_{i-1}$ fits within a single limb. This is the "else" branch of the "if" statement at line 10. This case is similar to the `getbit` function. We simply locate p_{i-n} and shift it to the least significant position, and then remove everything but the bottom n bits. Note that the `mod` computations at lines 3 and 12 are fast (a logical shift to compute the power of 2 minus one, and a bitwise `and` operation). Finally, in the third case, p_{i-n} is in one limb, and p_{i-1} is in the next one. Note that they cannot be more than one limb apart, as we require $n < 64$. In this case, at line 9 after shifting p_{i-n} to the least significant position, the bottom s bits of r are $p_{i-n} \dots p_{i-n+s-1}$, with $s < n$. The top $64 - s$ bits are zeroes. At line 11, we compute $p[k+1] \ll s$. Its bottom s bits are zeroes, and its top $64 - s$ bits are $p_{i-n+s} \dots p_{i-n+63}$. We sum this to r, and the result is $p_{i-n} \dots p_{i-n+63}$. At this point, all there is left to do is remove everything but the bottom n bits.

4.3.4 The main algorithm

We now have all the tools to describe the modular exponentiation algorithm proper. The `powm` function can be found in Alg. 15. It takes five pointers r, b, e, m and t, as parameters, as well as the lengths b_n, e_n, and n. Let $B = \texttt{value}(b, b_n)$, $E = \texttt{value}(e, e_n)$, and $M = \texttt{value}(m, n)$. It computes $B^E \bmod M$ and stores it in r. The extra buffer t is used as scratch space.

The core idea of the algorithm is Brauer 2^k-ary method [16]. Rather than considering E bit by bit, like in the usual square-and-multiply algorithm, we use a window of w bits, where w is a small number that depends on the length of e. It is chosen to optimize the number of multiplications and is always between 1 and 10.

Let us analyze the algorithm step by step.

Initialization (lines 2-4) The first thing the algorithm does is count the number of nontrivial bits in E. As a precondition requires $e[e_n - 1] > 0$, we know that there are less than 64 leading zeros. Therefore, the number of useful bits is $64e_n - k$, where k is the number of leading zeros in the limb $e[e_n - 1]$. There are various ways to count the leading zeroes of a machine integer. However, most compilers now provide efficient built-ins to do so. GMP uses the compiler built-in on many architectures. WhyMP uses the compiler built-in in all cases. We declare `count_leading_zeroes` as a `val` with the following specification. At extraction, it is replaced by calls to `__builtin_clzll`.

```
val count_leading_zeros (x:uint64) : int32
  requires { to_int x > 0 }
  ensures  { (power 2 (Int32.to_int result)) * to_int x ≤ max_uint64 }
  ensures  { 2 * (power 2 (Int32.to_int result)) * to_int x > max_uint64 }
  ensures  { 0 ≤ Int32.to_int result < 64 }
```

Once the number of bits in E is known, we can choose the window size w (line 3). The function `win_size` simply uses a lookup table to pick w between 1 and 10 depending on i. I did not bother reproducing it here, as the only important thing for the proof of correctness is that w is between 1 and 10.

We also need to compute the inverse of M modulo β in order to use the REDC algorithm. This is done at line 4 using the algorithm from Sec. 4.3.2.

Precomputing the odd powers of B (lines 5-13) The first step of the algorithm proper is to precompute a table of the odd powers of B up to $2^w - 1$ in Montgomery form. First, we allocate a pointer p of length $n \cdot 2^{w-1}$ (there are 2^{w-1} powers of B to store, and they have length n). We apply `redcify` (Alg. 11) to compute B in Montgomery form (line 7). At lines 8−9, we perform a multiplication in Montgomery form to compute B^2. As explained in Sec. 4.3.1,

Algorithm 15 Modular exponentiation.

```
Require: $\texttt{valid}(r,n) \wedge \texttt{valid}(b,b_n) \wedge \texttt{valid}(e,e_n) \wedge \texttt{valid}(m,n) \wedge \texttt{valid}(t,2n)$
Require: $\texttt{value}(m,n)$ odd
Require: $\texttt{value}(e,e_n) > 1$
Require: $e_n \geq 1 \wedge e[e_n - 1] > 0$
Require: $n \geq 1 \wedge m[n-1] > 0$
Require: $b_n \geq 1 \wedge b_n + n < 2^{32} - 1$          ▷ Precondition of REDCIFY.
Require: $512n \leq 2^{32}$                                ▷ Prevents overflow at line 12.
Require: $64(e_n + 1) < 2^{32} - 1$                        ▷ Prevents overflow at line 20.
Ensure: $\texttt{value}(r,n) \equiv \texttt{value}(b,b_n)^{\texttt{value}(e,e_n)} \mod \texttt{value}(m,n)$
Ensure: $0 \leq \texttt{value}(r,n) < \texttt{value}(m,n)$
 1: function POWM$(r,b,b_n,e,e_n,m,n,t)$
 2:     $i \leftarrow 64 \cdot e_n -$ COUNT_LEADING_ZEROS$(e[e_n - 1])$      ▷ $e$ is $i$ bits long.
 3:     $w \leftarrow$ WINDOW_SIZE$(i)$                          ▷ Length of the window.
 4:     $v \leftarrow -$BINVERT_LIMB$(m[0])$                     ▷ Let $E = \texttt{bitvalue}(e,i)$.
 5:     $p \leftarrow$ TMP_ALLOC$(n \cdot 2^{w-1})$              ▷ Let $B = \texttt{value}(b,b_n)$.
 6:     $p' \leftarrow p$                                        ▷ Let $M = \texttt{value}(m,n)$.
 7:     REDCIFY$(p,b,b_n,m,n)$                                   ▷ $\texttt{value}(p,n) \equiv \beta^n \cdot B \mod M$.
 8:     MUL_N$(t,p,p,n)$
 9:     REDC_1$(r,t,m,n,v)$                                      ▷ $\texttt{value}(r,n) \equiv \beta^n \cdot B^2 \mod M$.
10:     for $j = 1$ to $2^{w-1} - 1$ do    ▷ Precompute a table of the odd powers of $b$.
11:         MUL_N$(t,p',r,n)$
12:         $p' \leftarrow p' + n$
13:         REDC_1$(p',t,m,n,v)$                                 ▷ $\texttt{value}(p',n) \equiv \beta^n \cdot B^{2j+1} \mod M$.
14:     $x \leftarrow$ GETBITS$(e,i,w)$
15:     if $i < w$ then
16:         $i \leftarrow 0$
17:     else
18:         $i \leftarrow i - w$
19:     $c \leftarrow$ COUNT_TRAILING_ZEROS$(x)$                  ▷ $E = \texttt{bitvalue}(e,i) + 2^i x$.
20:     $i \leftarrow i + c$
21:     $x \leftarrow x \gg c$                                   ▷ $x$ is now odd.
22:     $h \leftarrow x \gg 1$                                   ▷ $x = 2h + 1$.
23:     COPYI$(r, p + n \cdot h, n)$                             ▷ $\texttt{value}(r,n) \equiv \beta^n \cdot B^x \mod M$.
24:     ghost $d \leftarrow x$                                   ▷ Invariant: $E = \texttt{bitvalue}(e,i) + 2^i d$.
25:     while $i > 0$ do                                         ▷ Invariant: $\texttt{value}(r,n) \equiv \beta^n \cdot B^d \mod M$.
26:         while GETBIT$(e,i) = 0$ do                           ▷ Sliding window optimization.
27:             MUL_N$(t,r,r,n)$
28:             REDC_1$(r,t,m,n,v)$
29:             $i \leftarrow i - 1$
30:             $d \leftarrow 2d$
31:             if $i = 0$ then
32:                 goto end
33:         $x \leftarrow$ GETBITS$(e,i,w)$
34:         if $i < w$ then
35:             $w' \leftarrow i$
36:         else
37:             $w' \leftarrow w$
38:         $i \leftarrow i - w'$                                ▷ $E = \texttt{bitvalue}(e,i) + 2^i(x + 2^{w'} d)$.
```

```
39:         $c \leftarrow$ COUNT_TRAILING_ZEROS$(x)$
40:         $i \leftarrow i + c$
41:         $x \leftarrow x \gg c$                                        ▷ $x$ is odd.
42:         $w' \leftarrow w' - c$
43:         while $w' > 0$ do      ▷ Invariant: $E = \texttt{bitvalue}(e, i) + 2^i(x + 2^{w'}d)$.
44:             MUL_N$(t, r, r, n)$
45:             REDC_1$(r, t, m, n, v)$
46:             $w' \leftarrow w' - 1$
47:             $d \leftarrow 2d$
                                            ▷ $E = \texttt{bitvalue}(e, i) + 2^i x + 2^i d$.
48:         $h \leftarrow x \gg 1$                                       ▷ $x = 2h + 1$.
49:         MUL_N$(t, r, p + n \cdot h, n)$
50:         REDC_1$(r, t, m, n, v)$
51:         $d \leftarrow d + x$
                                            ▷ $\texttt{value}(r, n) \equiv \beta^n \cdot B^E \mod M$
52: end:
53:     COPYI$(t, r, n)$
54:     ZERO$(t + n, n)$
55:     REDC_1$(r, t, m, n, v)$                 ▷ $\texttt{value}(r, n) \equiv B^E \mod M$
56:     if CMP$(r, m, n) \geq 0$ then           ▷ $M \leq \texttt{value}(r, n) < 2M$, adjust.
57:         SUB_N$(r, r, m, n)$
```

to perform a multiplication in Montgomery form, we first perform a regular multiplication with `mul_n`, and then divide by β^n using the REDC algorithm. This pattern occurs throughout the algorithm. The scratch buffer t is used to store the intermediary result of `mul_n`. The pointer r, which is meant to store the final result of the modular exponentiation, is used as temporary scratch space to store B^2 during the precomputation phase. Once we have B and B^2, the `for` loop at lines 10-13 computes all the remaining odd powers of B by repeatedly multiplying the last computed power of B by B^2.

More precisely, the loop invariants are as follows:

1) $p' = p + (j - 1) \cdot n$,

2) $\forall k.\ 0 \leq k < j \implies \texttt{value}(p + kn, n) \equiv \beta^n \cdot B^{2k+1} \mod M$.

First loop iteration, unrolled (lines 14-23) We extract the top w bits of E (line 14) and get some number x. We decrease i (lines 15-18) to reflect the remaining numbers of bits to consider in E. After the `if` block at line 18, the value of i has been updated and the following equality holds: $E = \texttt{bitvalue}(e, i) + 2^i x$. This is a direct consequence of the postcondition of `getbits` (Alg. 14).

The objective is now to compute B^x in Montgomery form. More precisely, let us decompose x as a product $2^c x'$ of some odd number x' and a power of 2. As x' is odd and smaller than 2^w, $B^{x'}$ is stored in the precomputed table p. So, we can compute B^x by getting $B^{x'}$ from the table and squaring it c times. The decomposition happens at line 19. The function `count_trailing_zeros` is the dual of `count_leading_zeros`. It is also a compiler built-in. The computation of $B^{x'}$ occurs at lines 21-23. We compute x' by shifting x to the right by c bits. We then compute $h = \lfloor x/2 \rfloor$ at line 22. From the invariant of the first loop, we get that $\texttt{value}(p + hn, n) \equiv \beta^n B^{x'}$, so we simply copy it over to r at line 23.

Now would be the time to square r c times to obtain B^x. However, GMP's implementation instead postpones this by increasing i by c at line 20. This puts back the c zero bits at the top of e, to be handled later on.

After line 23, the following equalities hold:

1) $\texttt{value}(r, n) = \beta^n \cdot B^{x'} \mod M$,

2) $E = \texttt{bitvalue}(e, i) + 2^i x'$.

Moreover, we know that $e_{i-c} \ldots e_{i-1}$ are all zeroes, although that is not really needed for our correctness proof. Finally, let us point out that this initial computation was actually an unrolled and optimized iteration of the main loop. The reason why it can be optimized is that we know that the top bit of e is a one after computing i.

The main loop (lines 25-51) This loop is the core of the algorithm. It consumes all the bits in e to compute $B^E \mod M$ in Montgomery form. Its invariants are best expressed using a ghost variable. We use the variable d to store the current exponent of B that is stored in r. Equivalently, d stores the bits of e that have already been handled. The fact that these two quantities are equal constitutes the loop invariant. More formally, the loop invariants are as follows:

1) $\texttt{value}(r, n) = \beta^n \cdot B^d \mod M$,

2) $E = \texttt{bitvalue}(e, i) + 2^i d$.

Of course, the invariants are initialized by setting d to x at line 24. As an aside, this is a very good use case for ghost code. It would be possible to express the invariants without making the variable d explicit, but they would be much harder to understand and it would result in harder goals for automated provers.

Let us now prove that these invariants are maintained through a loop iteration, and that i decreases at each loop iteration.

Sliding window optimization (lines 26-32) The first inner loop performs a small optimization. Rather than always handling blocks of w bits, we first absorb all the leading zeroes (and square r for each of them). The loop invariants of the inner loop are the same as those of the outer loop. They are obviously initialized, since the inner loop occurs at the very start of the outer loop iteration. It is easy to see that the invariants are maintained. At each iteration, d doubles and i is decreased by 1, while r is squared (in Montgomery form). If i becomes 0, we exit both this loop and the outer loop (and its invariants are valid).

The main loop, continued (lines 33-42) After the first inner loop, the situation is essentially the same as before (that is, the loop invariants are valid). We also know that the top bit of e is a one, which matters for the proof that the loop terminates. We call `getbits` to extract the top w bits of e (line 33). We store the number of bits that were actually extracted in a variable w' (lines 34-37). In every iteration except potentially the last one, $w = w'$. At line 38, we update i to reflect the new number of bits in e. At that point, the old value of i

is $i + w'$, so the following equality follows from the postcondition of `getbits`: $E = \texttt{bitvalue}(e, i) + 2^i x + 2^{i+w'} d$.

Much like in the unrolled iteration before the loop, we then split x into a product $2^c x'$ of a power of 2 and an odd number. Note that the top bit of x was known to be a one, so we know that $c < w'$. We put back the bottom c bits (which are all zeroes) on top of e, while increasing i by c and decreasing w' by c as well. We set $x = x'$. At this point, the equality $E = \texttt{bitvalue}(e, i) + 2^i x + 2^{i+w'} d$ still holds, although the values of i, x and w' have changed. However, as the top bit of e was a one, we know that i has decreased by at least one. This proves the termination of the loop.

Squaring (lines 43-47) At this point, r stores $B^d \mod M$ in Montgomery form. We need it to store $B^{x+2^{w'}d}$ at the end of the loop to validate the invariant. The loop at lines 43-47 computes $B^{2^{w'}d}$ (in Montgomery form) by squaring r w' times. More precisely, the invariants are as follows.

1) $\texttt{value}(r, n) \equiv \beta^n \cdot B^d \mod M$,

2) $E = \texttt{bitvalue}(e, i) + 2^i(x + 2^{w'} d)$.

At each iteration, d is doubled, w' is decreased by one, and r is squared (in Montgomery form), which ensures that the invariants are maintained.

The main loop, cont'd (lines 48-51) At line 48, the invariants of the last loop yield the following equality: $E = \texttt{bitvalue}(e, i) + 2^i(x + d)$. So, all there is to do is multiply r by B^x in Montgomery form. x is odd and less than 2^w, so we simply fetch B^x from the table p of precomputed values, much like in the unrolled loop iteration.

Final adjustments (lines 53-57) After the main loop, the invariants yield $\texttt{value}(r, n) \equiv \beta^n \cdot B^E \mod M$. So, we use the REDC algorithm (Alg. 10) to get it out of Montgomery form by dividing it by β^n. Since REDC requires a workspace of size $2n$, we copy r to the bottom n limbs of t and zero the top n limbs such that $\texttt{value}(t, 2n) = \texttt{value}(r, n)$. After the call to REDC at line 55, we have $\texttt{value}(r, n) \equiv B^E \mod M$, which fulfills the first postcondition. Moreover, since $\texttt{value}(r, n) < \beta^n$, in particular it is smaller than $\beta^n M$. Therefore, the second postcondition of `redc_1` yields $\texttt{value}(r, n) < 2M$. We need it smaller than M for the second postcondition, so we simply compare it to M (line 56) and subtract M from it if needed. Therefore, the first and third postcondition are fulfilled. The second postcondition is that r is unchanged except the range $[r, r+n)$. This is true because we never write anything into the cells that should be preserved (in the WhyML development, it is an additional invariant of all the loops in the function).

4.3.5 Side-channel resistance

A common use case for modular exponentiation is cryptography algorithms. A common concern for cryptography algorithms is side-channel resistance. Suppose the exponent e of modular exponentiation is a secret (such as a private key). Assume that the length of e is public knowledge (for example, it could be a property of the cryptographic protocol). Can we ensure that each execution of powm takes the same amount of time for all possible values of e with that length, and that the control flow is the same? Clearly, the algorithm from the previous section does not have this property. Indeed, the control flow is not always the same across calls to powm. The biggest offenders are the final adjustment step and the sliding window optimization. Moreover, the underlying primitives (add_n, redc_1, and so on) should be channel-resistant themselves, which is not the case.

GMP features another modular exponentiation algorithm called sec_powm. According to the documentation, it is channel-resistant, at the cost of some performance. I currently have no good way to use Why3 to prove the side-channel resistance of a program. However, it would be nice to at least verify the functional correctness of sec_powm, so as to increase the usefulness of WhyMP in practical contexts. Sadly, the verification of sec_powm is still a work in progress. The modular exponentiation itself is extremely similar to powm. However, there is a large number of nontrivial primitives to verify. In particular, a side-channel resistant division is required to implement redcify.

4.4 Toom-Cook multiplication

The schoolbook multiplication algorithm from Sec. 4.2.3 has quadratic complexity and is only optimal for numbers shorter than about 30 limbs, or 2000 bits[2]. For larger numbers (between 2000 and about 100 000 bits), GMP uses a family of recursive multiplication algorithms initially introduced by Toom [92] and Cook [23]. These algorithms for integer and polynomial multiplication can be viewed as solving a multipoint evaluation and polynomial interpolation problem.

The general principle of Toom-Cook algorithms is to choose a base B, typically a power of 2^{64}, and to view the digits of the factors in base B as coefficients of polynomials a and b. We then evaluate those polynomials at well-chosen points v_i, compute the products $a(v_i)b(v_i)$ by calling the algorithm recursively, and interpolate to obtain the coefficients of the product polynomial c. The product is then obtained by evaluating $c(B)$.

We have verified two Toom-Cook algorithms: Toom-2 (Sec. 4.4.1), which is similar to Karatsuba multiplication [55], and its unbalanced variant Toom-2.5 (Sec. 4.4.2), introduced by Bodrato and Zanoni [15].

Toom-2 (also called toom22 in the code) splits each of the operands into two parts roughly equal in length, and Toom-2.5 (or toom32) splits the largest operand into three parts and the smallest into two. Toom-2 is called on operands of roughly equal length and Toom-2.5 is called when one of the operands is about 1.5 times as long as the other. This way, after splitting, we are left with parts that have roughly equal length. A general case algorithm, which we describe in Sec. 4.4.3, reduces all cases to applications of the two former ones.

4.4.1 Toom-2

The parameters of the toom22_mul function (Alg. 16) are as follows: r is the destination buffer, a and b are the source operands, m and n are their lengths in limbs, and s is an extra buffer to store temporary results.

[2] The exact threshold depends on the compiler and architecture.

Algorithm 16 Toom-2 multiplication.

```
Require: $2 \le n \le m < 30 \cdot 2^k$
Require: $\texttt{valid}(r, m+n), \texttt{valid}(a, m), \texttt{valid}(b, n), \texttt{valid}(s, 2(m+k))$
Require: $4 \cdot m < 5 \cdot n$
Ensure: $\texttt{value}(r, m+n) = \texttt{value}(a, m) \cdot \texttt{value}(b, n)$
function TOOM22_MUL($r, a, b, s, m, n, k$)
    $\mu \leftarrow m \gg 1$
    $\lambda \leftarrow m - \mu$
    $\nu \leftarrow n - \lambda$
    $(a_0, a_1) \leftarrow (a, a + \lambda)$
    $(b_0, b_1) \leftarrow (b, b + \lambda)$
    Compute $|a(-1)|$ in $r$, $|b(-1)|$ in $r + \lambda$, sign in $\epsilon$ (see Alg. 18)
    $(c_0, c_\infty) \leftarrow (r, r + 2\lambda)$
    $s' \leftarrow s + 2\lambda$
    TOOM22_MUL_REC($s, r, r + \lambda, s', \lambda, \lambda, k - 1$)          ▷ Compute $|c(-1)|$.
    TOOM22_MUL_REC($c_\infty, a_1, b_1, s', \mu, \nu, k - 1$)          ▷ Compute $c(+\infty)$.
    TOOM22_MUL_REC($c_0, a_0, b_0, s', \lambda, \lambda, k - 1$)            ▷ Compute $c(0)$.
    $v \leftarrow$ ADD_N($c_\infty, c_0 + \lambda, c_\infty, \lambda$)                        ▷ $H_0 + L_\infty$.
    $v_2 \leftarrow v +$ ADD_N($c_0 + \lambda, c_\infty, c_0, \lambda$)                 ▷ $L_0 + H_0 + L_\infty$.
    $v \leftarrow v +$ ADD($c_\infty, c_\infty, \lambda, c_\infty + \lambda, \mu + \nu - \lambda$)          ▷ $H_0 + L_\infty + H_\infty$.
    if $\epsilon = -1$ then
        $v \leftarrow v +$ ADD($r + \lambda, r + \lambda, c_{-1}, 2\lambda$)
    else $v \leftarrow v -$ SUB($r + \lambda, r + \lambda, c_{-1}, 2\lambda$) mod $\beta$      ▷ $v \in \{0, 1, 2, \beta - 1\}$.
    INCR($c_\infty, v_2$)
    if $v \le 2$ then                                                  ▷ Implies $0 \le v$.
        INCR($r + 3\lambda, v$)
    else DECR($r + 3\lambda, 1$)              ▷ $v = \beta - 1$ instead of $-1$, due to integer
  representation.
```

Algorithm 17 Recursive call in Toom-2.

```
Require: $\texttt{valid}(r, m+n), \texttt{valid}(a, m), \texttt{valid}(b, n), \texttt{valid}(s, 2(m+k))$
Require: $0 < n \le m \le 2 \cdot n$
Ensure: $\texttt{value}(r, m+n) = \texttt{value}(a, m) \cdot \texttt{value}(b, n)$
  function TOOM22_MUL_REC($r, a, b, s, m, n, k$)
      if $n < 30$ then       ▷ Operands are small, use the schoolbook algorithm.
          MUL_BASECASE($r, a, b, m, n$)
      else
          if $4 \cdot m < 5 \cdot n$ then
              TOOM22_MUL($r, a, b, s, m, n, k$)
          else                          ▷ Operands are unbalanced, use Toom-2.5.
              TOOM32_MUL($r, a, b, s, m, n, k$)
```

Algorithm 18 Computation of $|a(-1)|$ in r and $|b(-1)|$ in $r+\lambda$.

```
Ensure: $\texttt{value}(r, \lambda) = |a(-1)|$
Ensure: $\texttt{value}(r + \lambda, \lambda) = |b(-1)|$
Ensure: $\epsilon \cdot \texttt{value}(r, \lambda) \cdot \texttt{value}(r + \lambda, \lambda) = a(-1)b(-1)$
  $\epsilon \leftarrow 1$                                    ▷ Will hold the sign of $a(-1)b(-1)$.
  if $\lambda = \mu$ then                                    ▷ Compute $a(-1)$.
      if COMPARE$(a_0, a_1, \lambda) < 0$ then               ▷ $A_1 > A_0$
          SUB_N$(r, a_1, a_0, \lambda)$
          $\epsilon \leftarrow -\epsilon$
      else SUB_N$(r, a_0, a_1, \lambda)$
  else                                                        ▷ $\lambda = \mu + 1$
      if $a_0[\mu] = 0 \wedge$ COMPARE$(a_0, a_1, \mu) < 0$ then   ▷ $A_1 > A_0$
          SUB_N$(r, a_1, a_0, \mu)$
          $r[\mu] \leftarrow 0$
          $\epsilon \leftarrow -\epsilon$
      else
          $t \leftarrow$ SUB_N$(r, a_0, a_1, \mu)$
          $r[\mu] \leftarrow a_0[\mu] - t$                   ▷ No borrow, as we know $a_0 \geq a_1$.
  if $\lambda = \nu$ then                                    ▷ Compute $b(-1)$.
      if COMPARE$(b_0, b_1, \lambda) < 0$ then               ▷ $B_1 > B_0$
          SUB_N$(r + \lambda, b_1, b_0, \lambda)$
          $\epsilon \leftarrow -\epsilon$                    ▷ Change the sign of $a(-1)b(-1)$.
      else SUB_N$(r + \lambda, b_0, b_1, \lambda)$
  else
      if IS_ZERO$(b_0 + \nu, \lambda - \nu) \wedge$ COMPARE$(b_0, b_1, \nu) < 0$ then   ▷ $B_1 > B_0$
          SUB_N$(r + \lambda, b_1, b_0, \nu)$                ▷ $b_0$ also has length at most $\nu$.
          ZERO$(r + \lambda + \nu, \lambda - \nu)$ ▷ We still have to initialize the rest of $(r + \lambda, \lambda)$.
          $\epsilon \leftarrow -\epsilon$
      else SUB$(r + \lambda, b_0, \lambda, b_1, \nu)$
```

The amount of space needed for s is approximately $2(m + \log_2(m))$ limbs. Rather than explicitly talking about logarithms in the specification, we use an extra parameter k. The variable k is ghost, which means that it is never used in the computations, but only in the proof. The first precondition ensures that k is a suitable bound. In practice, the caller of Toom-2 can give $k = 64$ (and allocate $2m + 128$ limbs as scratch space), as integer lengths are machine integers, so smaller than 2^{64}.

The last precondition makes sure that the operands are sufficiently close in size. It could be a bit looser without breaking the algorithm.

The algorithm is organized in four steps. First we split the operands into two parts of roughly equal length (Sec. 4.4.1). Then we evaluate the product polynomial c at three points (Sec. 4.4.1). We then recompose the coefficients of c through interpolation (Sec. 4.4.1). Finally, we propagate the remaining carries (Sec. 4.4.1).

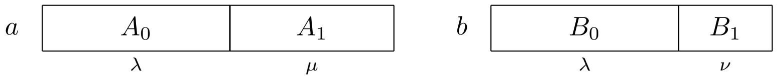
Splitting (Alg. 16, lines 2-6)

We pose $\mu = \lfloor \frac{m}{2} \rfloor$, $\lambda = m - \mu$, $\nu = n - \lambda$. The preconditions ensure the following:

$$\begin{aligned} 0 < \nu \leq \mu \leq \lambda < m, \\ \lambda - 1 \leq \mu \leq \lambda, \\ \lambda < \mu + \nu. \end{aligned}$$

We can split a into two subwords a_0 and a_1 such that $\texttt{value}(a, m)$ equals $\texttt{value}(a_0, \lambda) + \beta^\lambda \cdot \texttt{value}(a_1, \mu)$. Similarly we have b_0 and b_1 such that $\texttt{value}(b, n)$ equals $\texttt{value}(b_0, \lambda) + \beta^\lambda \cdot \texttt{value}(b_1, \nu)$.

We denote $A_0 := \texttt{value}(a_0, \lambda)$ and so on. Likewise, we define the polynomials $a(X) := A_0 + A_1 X$ and $b(X) := B_0 + B_1 X$. The goal is to compute $c(\beta^\lambda)$, where $c(X) = a(X)b(X)$ is a degree-2 polynomial. We pose $c(X) = C_0 + C_1 X + C_2 X^2$.



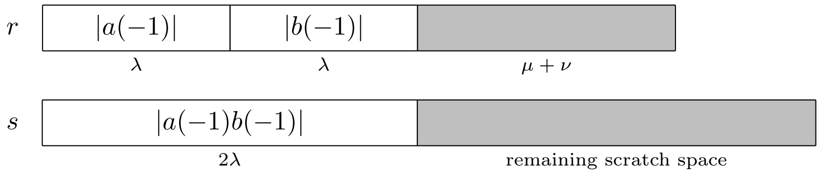
Evaluation (Alg. 16, lines 7-12)

The first step is to obtain three values of $c(X)$. GMP chooses to evaluate c at 0, -1, and $+\infty$ (where $c(+\infty)$ is defined as C_2). We first evaluate $a(-1) = A_0 - A_1$ and $b(-1) = B_0 - B_1$ (Alg. 18). To avoid carry propagations, we first check which of A_0 and A_1 is larger to compute $|a(-1)|$ and store its sign separately in a variable ϵ. If $\mu = \lambda - 1$, we can optimize slightly by performing a subtraction of length μ instead of λ.

Similarly, we compute $|b(-1)|$ and update ϵ to contain the sign of $a(-1)b(-1)$. We store $|a(-1)|$ in the first λ limbs of r (we denote this subarray $r(0, \lambda)$) and $|b(-1)|$ in the next λ limbs (which we denote $r(\lambda, 2\lambda)$). We then call Toom-2 recursively to compute $|a(-1)b(-1)|$ and store the result in $s(0, 2\lambda)$ (Alg. 16, line 10). We use $s(2\lambda, \ldots)$ as scratch space (there is enough space because k decreased by one).

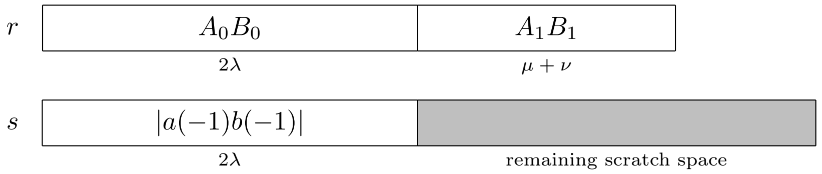
The constant 30 in the recursive call function `toom22_mul_rec` (Alg. 17) is the minimum integer length for which our library calls Toom-Cook multiplication algorithms. For smaller numbers, the schoolbook multiplication is called instead. The value 30 was picked experimentally, and the optimum can vary from machine to machine. What matters is that the value is high enough that the preconditions of `toom22_mul` and `toom32_mul` are respected. The relevant precondition of `toom32_mul` is $n + 2 \leq m \wedge m + 6 \leq 3 \cdot n$ (the full specification can be found in Sec. 4.4.2). Provided the threshold is above 8, the precondition is met.

After the recursive call at line 10, the memory layout is as follows. Throughout this section, the memory layout will be illustrated with such diagrams (in addition to the formulas) to help visualize the current state of the computation.



The next step is to compute $c(+\infty)$, that is, A_1B_1. This is done with a simple recursive call to Toom-2 (Alg. 16, line 11), using $s(2\lambda, \ldots)$ as scratch space again. The result has size $\mu + \nu$, which fits in $r(2\lambda, 2\lambda + \mu + \nu)$.

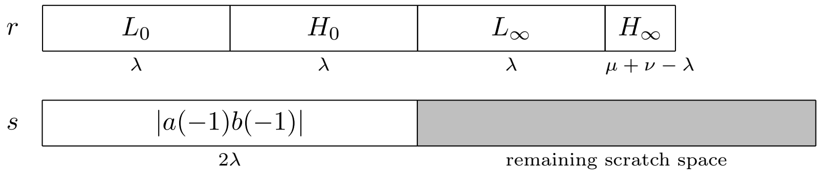
Finally, we compute $c(0)$, that is, A_0B_0 (line 12). We use the second half of s as scratch space again, and store the result in $r(0, 2\lambda)$, writing over $|a(-1)|$ and $|b(-1)|$ (but their product still is in the first half of s).



We further decompose $c(0)$, $c(-1)$, and $c(+\infty)$ in halves of size λ or less:

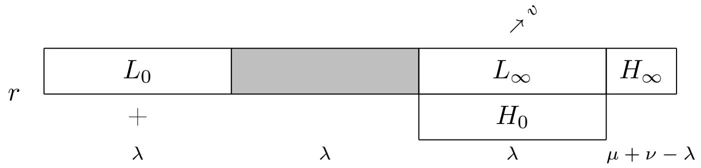
$$A_0B_0 = c(0) = L_0 + \beta^\lambda \cdot H_0$$
$$A_1B_1 = c(+\infty) = L_\infty + \beta^\lambda \cdot H_\infty.$$

At the end of the evaluation step, we have the following memory layout:

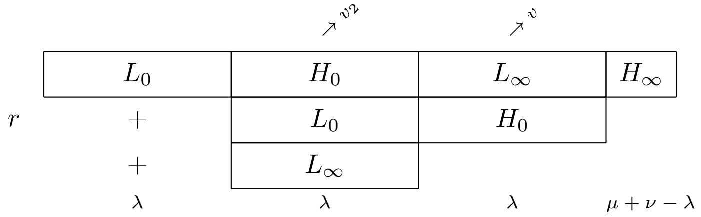


Recomposition (Alg. 16, lines 13-18)

We first add H_0 to L_∞ in place, storing the carry in a variable v (line 13). The outgoing up-right arrows in the following diagrams represent the carry out.

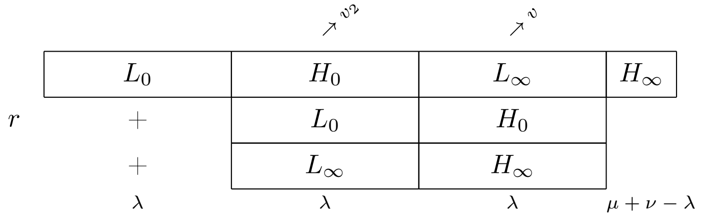


We then add the result to L_0, writing over the original location of H_0 (line 14). We store the sum of v and the new carry in v_2. At that point, $\texttt{value}(r+\lambda,\lambda)+\beta^\lambda v_2 = L_0+H_0+L_\infty$.



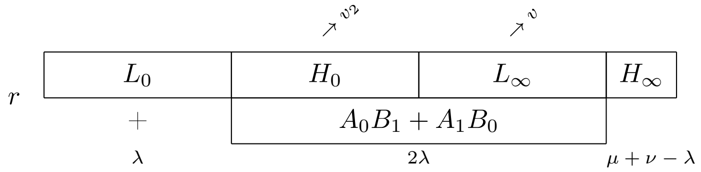
We add H_∞ to $r(2\lambda, 3\lambda)$ in place, incrementing v if needed (line 15). At that point, we have:

$$\begin{aligned}
&\texttt{value}(r+\lambda,2\lambda)+\beta^\lambda v_2+\beta^{2\lambda}v\\
&=(H_0+L_0+L_\infty)+\beta^\lambda(H_0+L_\infty+H_\infty)\\
&=A_0B_0+A_1B_1+H_0+\beta^\lambda L_\infty. \qquad (4.1)
\end{aligned}$$



Finally, we subtract $c(-1)$ from $r(\lambda, 3\lambda)$ by adding or subtracting $|a(-1)b(-1)|$, depending on the stored sign (lines 16-18). At that point, we have $-1 \le v \le 3$, and:

$$\begin{aligned}
&\texttt{value}(r+\lambda,2\lambda)+\beta^\lambda v_2+\beta^{2\lambda}v\\
&=A_0B_0+A_1B_1+H_0+\beta^\lambda L_\infty-(A_0-A_1)(B_0-B_1)\\
&=A_0B_1+A_1B_0+H_0+\beta^\lambda L_\infty.
\end{aligned}$$



Therefore,

$$\begin{aligned}&\texttt{value}(r, m+n) + \beta^{2\lambda}v_2 + \beta^{3\lambda}v \\ &= L_0 + \beta^{\lambda}(\texttt{value}(r+\lambda, 2\lambda) + \beta^{\lambda}v_2 + \beta^{2\lambda}v) + \beta^{3\lambda}H_\infty \\ &= A_0B_0 + \beta^{\lambda}(A_0B_1 + A_1B_0) + \beta^{2\lambda}A_1B_1 \\ &= (A_0 + \beta^{\lambda}A_1)(B_0 + \beta^{\lambda}B_1).\end{aligned}$$

The only thing left to do is propagating the carries.

Carry propagation (Alg. 16, lines 19-22)

We first propagate v_2 at line 19, then v (lines 20-22). There is an `if` statement because the case $v_2 = -1$ (represented as the unsigned integer $\beta - 1$) requires special treatment, as $\texttt{incr}(\cdot, \beta-1)$ is not the same thing as $\texttt{decr}(\cdot, 1)$. Indeed, when propagating the carries, the functions `incr` and `decr` never check whether they reach the bounds of the array that holds the number to be incremented. Rather, they perform an addition (or subtraction) and propagate the carry until there is none. Their preconditions include the fact that this computation should not overflow. Calling these functions incorrectly could result in buffer overflows. Notably, this is a situation where the memory safety of the program (absence of buffer overflow) directly depends on its functional correctness (the number that is being incremented fits between certain bounds).

The most difficult part of the whole Toom-2 proof is ensuring that the propagation of the carries v and v_2 does not overflow. This is easy when $v \geq 0$, as the total product ab (obtained after propagating both carries) is certain to fit in $m + n$ limbs and the intermediate value obtained after propagating one of the carries is certain to be between 0 and ab. The only nontrivial case is $v = -1, v_2 \neq 0$. There, it is not obvious that the propagation of v_2 does not overflow out of r, as we have:

$$\texttt{value}(r, m+n) = (A_0 + \beta^{\lambda}A_1)(B_0 + \beta^{\lambda}B_1) - \beta^{2\lambda}v_2 + \beta^{3\lambda},$$

and we might have $(A_0 + \beta^{\lambda}A_1)(B_0 + \beta^{\lambda}B_1) + \beta^{3\lambda} \geq \beta^{m+n}$.

It is sufficient to show that $H_\infty < \beta^{\mu+\nu-\lambda} - 1$, that is, that the binary representation of H_∞ is not all ones. Indeed, the carry is absorbed somewhere in H_∞ if it is the case. If $v = -1$, the first addition (of H_0 and L_∞) cannot overflow, and we must have $v_2 \leq 1$. The only case to consider is therefore $v = -1$, $v_2 = 1$. If $v = -1$, the subtraction of c_{-1} from $r + \lambda$ necessarily underflows, that is,

$$A_0B_0 + A_1B_1 + H_0 + \beta^{\lambda}L_\infty - \beta^{\lambda} < (A_0 - A_1)(B_0 - B_1)$$

(Equation (4.1) with $v_2 = 1$, $v = 0$).

Noticing that $0 \leq H_0$ and $0 \leq A_0B_1 + A_1B_0 = A_0B_0 + A_1B_1 - (A_0 - A_1)(B_0 - B_1)$, we are left with $\beta^{\lambda}L_\infty - \beta^{\lambda} < 0$, which implies $L_\infty = 0$. Let us pose x, y the 2-adic valuations of A_1 and B_1, and a', b' odd integers such that $A_1 = 2^x a'$, $B_1 = 2^y b'$. We have $2^{x+y}a'b' = A_1B_1 = \beta^{\lambda}H_\infty = 2^{64\lambda}H_\infty$. As a' and b' are odd we must have $x + y \geq 64\lambda$. If H_∞ is even we are done (its binary representation cannot be all ones), so we can assume $x + y = 64\lambda$ and $a'b' = H_\infty$ without loss of generality. Notice now that $A_1 < 2^{64\mu}$ as it fits in a zone of length μ. We therefore have $x < 64\mu$ and $a' \leq 2^{64\mu - x} - 1$, similarly $y < 64\nu$ and $b' \leq 2^{64\nu - y} - 1$. Therefore,

$$\begin{aligned} H_\infty = a'b' &\leq (2^{64\mu - x} - 1)(2^{64\nu - y} - 1) \\ &= \beta^{\mu+\nu-\lambda} - 2^{64\nu - y} - 2^{64\mu - x} + 1 \\ &\leq \beta^{\mu+\nu-\lambda} - 3, \end{aligned}$$

and we can conclude that the propagation of v_2 does not overflow.

We did not expect this fact to require such a complex proof, especially because there were no comments in the GMP source code to indicate that nontrivial reasoning was needed. We discussed this with the developers, and they ended up changing the algorithm to make it more clearly correct at no performance cost.[3]

4.4.2 Toom-2.5

Toom-2.5, also known as `toom32`, has the same signature as Toom-2. The algorithm is similar (Alg. 19), but splits the larger operand into three parts rather than two. It is optimally used when the first operand is longer than the second by half.

The algorithm is organized into similar phases as Toom-2. First we split the larger operand into three parts and the smaller one into two parts. Then we evaluate c at four points, and finally we recompose the coefficients of c.

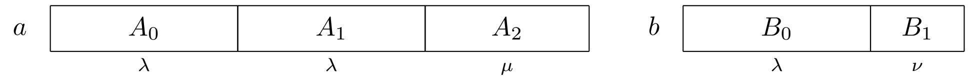
Splitting (Alg. 19, lines 2-9)

We pose either $\lambda = 1 + \lfloor \frac{m-1}{3} \rfloor$ (that is, $\lceil m/3 \rceil$) or $\lambda = 1 + \lfloor \frac{n-1}{2} \rfloor$ (or $\lceil n/2 \rceil$), whichever is largest. We also pose $\mu = m - 2\lambda$ and $\nu = n - \lambda$. The preconditions ensure that $0 < \nu \leq \lambda$, $0 < \mu \leq \lambda$, and $\lambda \leq \mu + \nu$.

Similarly to Toom-2, we split a into three subwords a_0, a_1 and a_2 of lengths λ, λ, and μ, respectively. We also split b into two subwords b_0 and b_1 of lengths λ and ν. We denote $A_0 := \texttt{value}(a_0, \lambda)$ and so on. We define the polynomials $a(X) := A_0 + A_1 X + A_2 X^2$ and $b(X) := B_0 + B_1 X$. The goal is to compute $c(\beta^\lambda)$, where $c(X) = a(X)b(X)$ is a degree-3 polynomial. We pose $c(X) = C_0 + C_1 X + C_2 X^2 + C_3 X^3$, and we have the following:

$$\begin{aligned} C_0 &= A_0 B_0, \\ C_1 &= A_1 B_0 + A_0 B_1, \\ C_2 &= A_2 B_0 + A_1 B_1, \\ C_3 &= A_2 B_1. \end{aligned}$$

The lengths of A_i and B_i coefficients imply that C_0 has length 2λ, C_1 and C_2 have length $2\lambda + 1$ and C_3 has length $\mu + \nu$.



[3] https://gmplib.org/repo/gmp/rev/02a2ec6e1bce

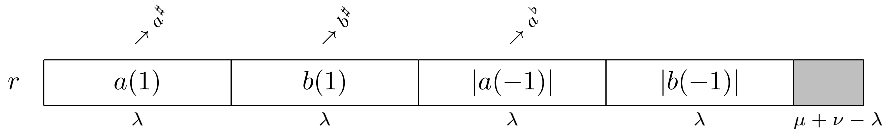
Algorithm 19 Toom-2.5 multiplication.

```
Require: $30 \le n \le m < 30 \cdot 2^k$
Require: $\mathtt{valid}(r, m+n), \mathtt{valid}(a, m), \mathtt{valid}(b, n), \mathtt{valid}(s, 2(m+k))$
Require: $n + 2 \le m \wedge m + 6 \le 3 \cdot n$
Ensure: $\mathtt{value}(r, m+n) = \mathtt{value}(a, m) \cdot \mathtt{value}(b, n)$
function TOOM32_MUL$(r, a, b, s, m, n, k)$
    if $2m \ge 3n$ then
        $\lambda \leftarrow 1 + \lfloor \frac{m-1}{3} \rfloor$
    else
        $\lambda \leftarrow 1 + \lfloor \frac{n-1}{2} \rfloor$
    $\mu \leftarrow m - 2\lambda$
    $\nu \leftarrow n - \lambda$
    $(a_0, a_1, a_2) \leftarrow (a, a + \lambda, a + 2\lambda)$
    $(b_0, b_1) \leftarrow (b, b + \lambda)$
     Compute $a(1)$, $b(1)$, $|a(-1)|$, $|b(-1)|$ in $r$, sign in $\epsilon$ (see Alg. 20).
     Carries of $a(1)$ and $b(1)$ are in $a^\sharp$ and $b^\sharp$, carry of $|a(-1)|$ in $a^\flat$.
    $s' \leftarrow s + 2\lambda + 1$
    TOOM22_MUL_REC$(s, r, r + \lambda, s', \lambda, \lambda, k - 1)$    ▷ Compute $c(1)$ recursively.
    if $a^\sharp = 1$ then    ▷ Propagate the carry for $c(1)$.
        $v \leftarrow b^\sharp +$ ADD_N$(s + \lambda, s + \lambda, r + \lambda, \lambda)$
    else
        if $a^\sharp = 2$ then
            $v \leftarrow 2b^\sharp +$ ADDMUL_1$(s + \lambda, r + \lambda, \lambda, 2)$
        else    ▷ $a^\sharp = 0$
            $v \leftarrow 0$
    if $b^\sharp$ then
        $v \leftarrow v +$ ADD_N$(s + \lambda, s + \lambda, r, \lambda)$
    $s[2\lambda] \leftarrow v$
    TOOM22_MUL_REC$(r, r + 2\lambda, r + 3\lambda, s', \lambda, \lambda, k - 1)$    ▷ Compute $|c(-1)|$ recursively.
    if $a^\flat$ then    ▷ Propagate the carry for $|c(-1)|$.
        $a^\flat \leftarrow$ ADD_N$(r + \lambda, r + \lambda, r + 3\lambda, \lambda)$
    $r[2\lambda] \leftarrow a^\flat$
    if $\epsilon = -1$ then
        SUB_N$(s, s, r, 2\lambda + 1)$
    else
        ADD_N$(s, s, r, 2\lambda + 1)$
    RSHIFT$(s, s, 2\lambda + 1, 1)$    ▷ $s \leftarrow \frac{c(1)+c(-1)}{2} = C_0 + C_2$
    $v \leftarrow$ ADD_N$(r + 2\lambda, s, s + \lambda, \lambda)$    ▷ Add $L$ and $M$ in $D_1$.
    INCR$(s + \lambda, \lambda + 1, v + s[2\lambda])$    ▷ Propagate $v$ and $h$ to $D_2$.
    if $\epsilon = -1$ then
        $v \leftarrow$ ADD_N$(s, s, r, \lambda)$
        $v' \leftarrow r[2\lambda] +$ ADD_NC$(r + 2\lambda, r + 2\lambda, r + \lambda, \lambda, v)$
        INCR$(s + \lambda, \lambda + 1, v')$
```

```
37:  else
38:      $v \leftarrow$ SUB_N$(s, s, r, \lambda)$
39:      $v' \leftarrow r[2\lambda] +$ SUB_NC$(r + 2\lambda, r + 2\lambda, r + \lambda, \lambda, v)$
40:      DECR$(s + \lambda, \lambda + 1, v')$
41:  TOOM22_MUL_REC$(r, a_0, b_0, s', \lambda, \lambda, k - 1)$ ▷ Compute $c(0)$ recursively.
42:  if $\mu > \nu$ then                                  ▷ Compute $c(+\infty)$ recursively.
43:      MUL$(r + 3\lambda, a_2, \mu, b_1, \nu)$
44:  else
45:      MUL$(r + 3\lambda, b_1, \nu, a_2, \mu)$
46:  $v \leftarrow$ SUB_N$(r + \lambda, r + \lambda, r + 3\lambda, \lambda)$
47:  $v' \leftarrow s[2\lambda] + v$
48:  $v \leftarrow$ SUB_NC$(r + 2\lambda, r + 2\lambda, r, \lambda, v)$
49:  $v' \leftarrow v' -$ SUB_NC$(r + 3\lambda, s + \lambda, r + \lambda, \lambda, v)$
50:  $v' \leftarrow v' +$ ADD$(r + \lambda, r + \lambda, 3\lambda, s, \lambda)$
51:  if $\mu + \nu > \lambda$ then                             ▷ Propagate $v'$.
52:      $v' \leftarrow v' -$ SUB$(r + 2\lambda, r + 2\lambda, 2\lambda, r + 4\lambda, \mu + \nu - \lambda)$
53:      if $v' < 0$ then
54:          DECR$(r + 4\lambda, \mu + \nu - \lambda, -v')$
55:      else
56:          INCR$(r + 4\lambda, \mu + \nu - \lambda, v')$
```

Evaluation in 1 and -1 (Alg. 19, lines 10-25, and full Alg. 20)

As $c(X)$ has degree 3, this time we need to obtain four values for interpolation. GMP chooses to evaluate c at $0, -1, 1$ and $+\infty$. We remark that $a(-1) = A_0 - A_1 + A_2$ and $a(1) = A_0 + A_1 + A_2$. Therefore, some evaluation steps can be saved by computing $A_0 + A_2$ only once and using this result for both evaluations (Alg. 20). Just like in Toom-2, we actually compute $|a(-1)|$ and $|b(-1)|$ and store the sign of the product separately in a variable ϵ. The carries of the evaluation of $a(1)$, $b(1)$ and $|a(-1)|$ are stored in $a^\sharp$, $b^\sharp$ and $a^\flat$ respectively. The evaluation of $|b(-1)| = |B_0 - B_1|$ cannot overflow (and this is why we bother computing absolute values and storing the sign separately).



After this step, we have $a(1) = \texttt{value}(r, \lambda) + \beta^\lambda a^\sharp$ and so on. We also have $0 \leq a^\sharp \leq 2$, $0 \leq b^\sharp \leq 1$, and $0 \leq a^\flat \leq 1$. The next thing to do is to compute $c(1)$.

$$c(1) = a(1)b(1) = (\texttt{value}(r, \lambda) + \beta^\lambda a^\sharp) \cdot (\texttt{value}(r + \lambda, \lambda) + \beta^\lambda b^\sharp)$$

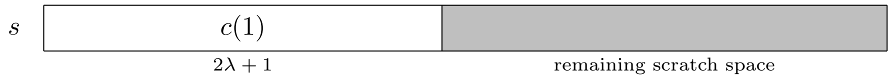
The recursive call at line 11 of Alg. 19 computes $\texttt{value}(r, \lambda) \cdot \texttt{value}(r + \lambda, \lambda)$ in s. The `if` statement at line 12 adds $a^\sharp \texttt{value}(r + \lambda, \lambda)$, shifted by λ as this term must be multiplied by β^λ. It also accumulates the carry and $a^\sharp b^\sharp$ into the variable v. After this, we have

$$\texttt{value}(s, 2\lambda) + \beta^{2\lambda} v = (\texttt{value}(r, \lambda) + \beta^\lambda a^\sharp) \cdot \texttt{value}(r + \lambda, \lambda) + \beta^{2\lambda} a^\sharp b^\sharp$$

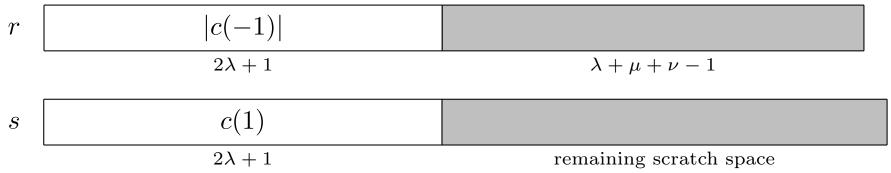
Algorithm 20 Toom-2.5: evaluation in 1 and -1.

```
Ensure: $\texttt{value}(r, \lambda) + \beta^\lambda a^\sharp = a(1)$
Ensure: $\texttt{value}(r + \lambda, \lambda) + \beta^\lambda b^\sharp = b(1)$
Ensure: $\texttt{value}(r + 2\lambda, \lambda) + \beta^\lambda a^\flat = |a(-1)|$
Ensure: $\texttt{value}(r + 3\lambda, \lambda) = |b(-1)|$
Ensure: $\epsilon \cdot (\texttt{value}(r + 2\lambda, \lambda) + \beta^\lambda a^\flat) \cdot \texttt{value}(r + 3\lambda, \lambda) = a(-1)b(-1)$
  $a' \leftarrow \textsc{add}(r, a_0, \lambda, a_2, \mu)$
  if $a' = 0 \wedge \textsc{cmp}(r, a_1, \lambda) < 0$ then          ▷ $A_0 + A_2 < A_1$
      $\textsc{sub\_n}(r + 2\lambda, a_1, r, \lambda)$
      $a^\flat \leftarrow 0$
      $\epsilon \leftarrow -1$
  else                                                                ▷ $A_1 \leq A_0 + A_2$
      $v \leftarrow \textsc{sub\_n}(r + 2\lambda, r, a_1, \lambda)$
      $a^\flat \leftarrow a' - v$
      $\epsilon \leftarrow 1$
  $a^\sharp \leftarrow a' + \textsc{add\_n}(r, r, a_1, \lambda)$          ▷ Finish computing $a(1)$.
  if $\lambda = \nu$ then
      $b^\sharp \leftarrow \textsc{add\_n}(r + \lambda, b_0, b_1, \lambda)$
      if $\textsc{cmp}(b_0, b_1, \lambda) < 0$ then                      ▷ $B_0 < B_1$
          $\textsc{sub\_n}(r + 3\lambda, b_1, b_0, \lambda)$
          $\epsilon \leftarrow -\epsilon$
      else
          $\textsc{sub\_n}(r + 3\lambda, b_0, b_1, \lambda)$
  else
      $b^\sharp \leftarrow \textsc{add}(r + \lambda, b_0, \lambda, b_1, \nu)$
      if $\textsc{is\_zero}(b_0 + \nu, \lambda - \nu) \wedge \textsc{cmp}(b_0, b_1, \nu) < 0$ then   ▷ $B_0 < B_1$
          $\textsc{sub\_n}(r + 3\lambda, b_1, b_0, \nu)$          ▷ $b_0$ also has length at most $\nu$.
          $\textsc{zero}(r + 3\lambda + \nu, \lambda - \nu)$        ▷ We still have to initialize the rest of $(r + 3\lambda, \lambda)$.
          $\epsilon \leftarrow -\epsilon$
      else
          $\textsc{sub}(r + 3\lambda, b_0, \lambda, b_1, \nu)$
```

The only missing term is $\beta^\lambda b^\sharp \cdot \texttt{value}(r, \lambda)$, which is handled in the `if` statement at line 16 of Alg. 19. After this, we have $\texttt{value}(s, 2\lambda) + \beta^{2\lambda} v = c(1)$, and we can set $s[2\lambda]$ to v.



The product $|c(-1)| = |a(-1)b(-1)|$ is computed similarly at lines 22-25. As there is no carry corresponding to $|b(-1)|$, the procedure is a bit simpler. The term $\beta^\lambda a^\flat \cdot \texttt{value}(r + 3\lambda, \lambda)$ is added in the `if` statement at line 23. The product is stored in $r(0, 2\lambda + 1)$. This overwrites $a(1)$, $b(1)$ and part of $|a(-1)|$, but these intermediate results are no longer needed.



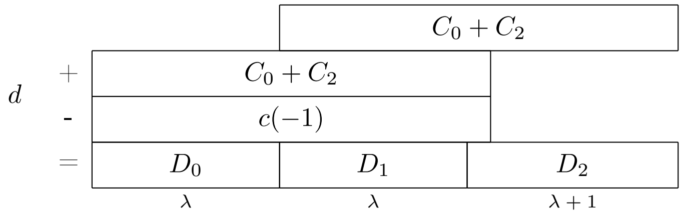
Recomposition (Alg. 19, lines 26-40)

We use the intermediate variable $d := C_1 + C_3 + \beta^\lambda \cdot (C_0 + C_2)$. The sizes of the C_i imply that d has length $3\lambda + 1$, so we pose D_0, D_1, D_2 such that $d = D_0 + \beta^\lambda D_1 + \beta^{2\lambda} D_2$ with D_0 and D_1 of length λ and D_2 of length $\lambda + 1$.

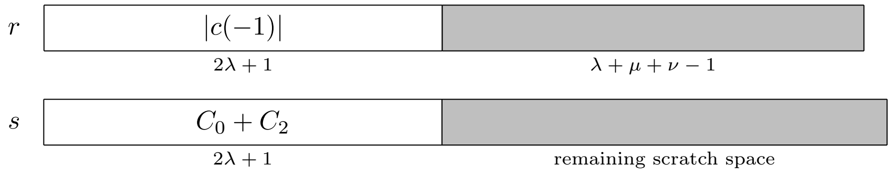
Note that $C_1 + C_3 = C_0 + C_2 - (C_0 - C_1 + C_2 - C_3) = C_0 + C_2 - c(-1)$, so:

$$d = C_0 + C_2 - c(-1) + \beta^\lambda \cdot (C_0 + C_2).$$

The computation of d goes as follows.

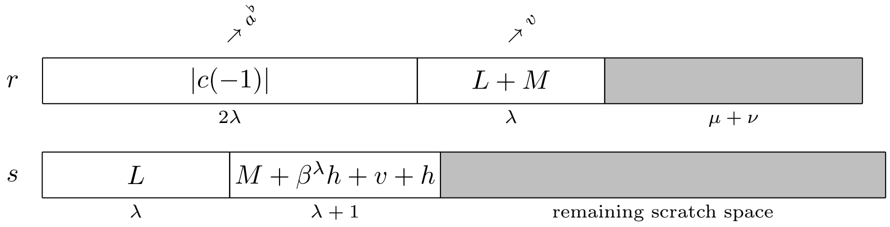


We compute D_0 at s, D_1 at $r + 2\lambda$ and D_2 at $s + \lambda$. The first step is to write $C_0 + C_2$ in s. Noticing that $c(1) + c(-1) = 2 \cdot (C_0 + C_2)$, this is easy to do with one long addition or subtraction (depending on ϵ) of r and s (lines 26-29), and then a logical shift on the result to do the division by 2 (line 30).



Let us note h the highest limb of $C_0 + C_2$ and split the 2λ remaining limbs into two subwords L and M, such that $C_0 + C_2 = L + \beta^\lambda M + \beta^{2\lambda} h$.

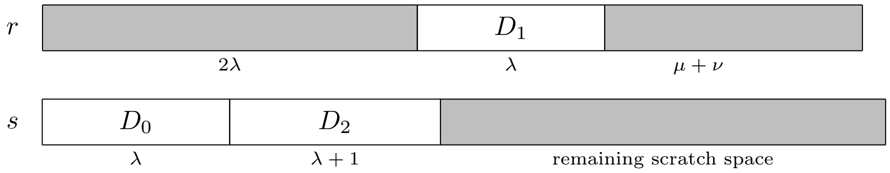
We add L and M together and write the result in $r + 2\lambda, \lambda$, the location of D_1 (line 31 Alg. 19). This overwrites the most significant limb of $|c(-1)|$, but it is still stored in $a^\flat$. The carry v is propagated and added to D_2, or $s + \lambda$, which already contains the higher half of $C_0 + C_2$ (line 32).



The only thing left to do in the computation of d is to subtract $c(-1)$. The `if` statement at line 33 does so (by adding or subtracting $|c(-1)|$ depending on ϵ). As D_0 and D_1 are not stored contiguously, we have to subtract both halves of $c(-1)$ separately. The `add_nc` and `sub_nc` functions are variants of addition and subtraction that take a carry input to add or subtract to the result, they allow to propagate the carry from the subtraction of the lower half of $c(-1)$ to the upper half. The carry of the upper half subtraction and $a^\flat$ (the high limb of $|c(-1)|$) are propagated to D_2.

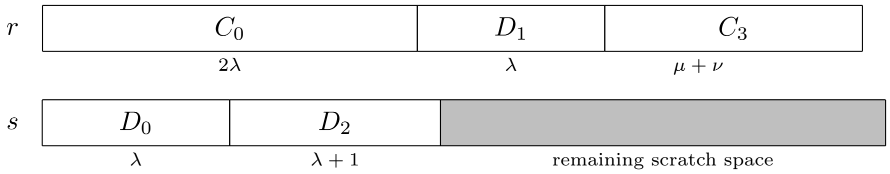
At that point, we can check the following:

$$\begin{aligned}
&\texttt{value}(s, \lambda) + \beta^\lambda \texttt{value}(r + 2\lambda, \lambda) + \beta^{2\lambda} \texttt{value}(s + \lambda, \lambda + 1) \\
&= L + \beta^\lambda (L + M - \beta^\lambda v) + \beta^{2\lambda}(M + \beta^\lambda h + v + h) - c(-1) \\
&= L + \beta^\lambda M + \beta^{2\lambda} h + \beta^\lambda (L + \beta^\lambda M + \beta^{2\lambda} h) - c(-1) \\
&= C_0 + C_2 + \beta^\lambda (C_0 + C_2) - c(-1) \\
&= d.
\end{aligned}$$



Evaluation in 0 and $+\infty$ (Alg. 19, lines 41-45)

We recursively compute C_0 in $(r, 2\lambda)$ and C_3 in $(r + 3\lambda, \mu + \nu)$. The upper half of s is still available and is used as scratch space. As the operands can be very unbalanced in the case of C_3, we have used the generic multiplication (see Sec. 4.4.3) instead of calling Toom-2 or Toom-2.5 directly.



Recomposition (Alg. 19, lines 46-56)

Let us note L_0 and H_0 the two halves of C_0, and split $C_3 = c(\infty)$ into L_∞ and H_∞ of length λ and $\mu + \nu - \lambda$ respectively. Note that H_∞ can have length 0.

$$\begin{aligned} C_0 &= L_0 + \beta^\lambda H_0 \\ C_3 &= L_\infty + \beta^\lambda H_\infty \end{aligned}$$

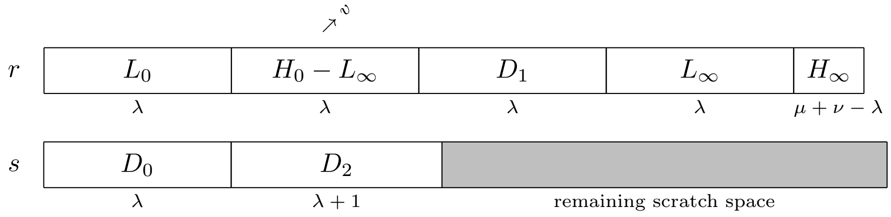
The product $c(\beta^\lambda) = a(\beta^\lambda)b(\beta^\lambda)$ can be expressed only in terms of C_0, C_3 and d:

$$\begin{aligned} &C_0 + \beta^\lambda d + \beta^{3\lambda} C_3 - C_0\beta^{2\lambda} - \beta^\lambda C_3 \\ &= C_0 + \beta^\lambda(C_1 + C_3) + \beta^{2\lambda}(C_0 + C_2) + \beta^{3\lambda}C_3 - C_0\beta^{2\lambda} - \beta^\lambda C_3 \\ &= C_0 + \beta^\lambda C_1 + \beta^{2\lambda}C_2 + \beta^{3\lambda}C_3 \\ &= c(\beta^\lambda). \end{aligned}$$

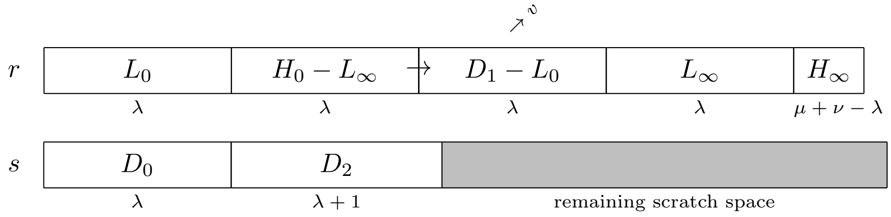
This implies the following decomposition for $c(\beta^\lambda)$:

$$\begin{aligned} c(\beta^\lambda) &= C_0 + \beta^\lambda d + \beta^{3\lambda}C_3 - C_0\beta^{2\lambda} - \beta^\lambda C_3 \\ &= L_0 + \beta^\lambda(D_0 + (H_0 - L_\infty)) + \beta^{2\lambda}(D_1 - L_0 - H_\infty) + \beta^{3\lambda}(D_2 - (H_0 - L_\infty)) + \beta^{4\lambda}H_\infty. \end{aligned}$$

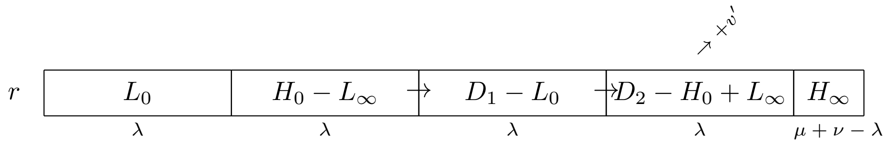
The first step is to subtract L_∞ from H_0 at $r + \lambda$ (lines 46-47). The borrow is stored in v. The variable v' contains the sum of v and the high limb of D_2.



The next step is to subtract L_0 from $r + 2\lambda$ (line 48). The borrom v from the previous computation at $r + \lambda$ is also propagated into $r + 2\lambda$. The final borrow is stored in v.



We then subtract $H_0 - L_\infty$ from D_2 at $r + 3\lambda$, propagating the previous carry (line 49). The carry out is accumulated in v'.



s	D_0	D_2	
	λ	$\lambda + 1$	remaining scratch space

We add D_0 at $r + \lambda$, and propagate the carry all the way to $r + 4\lambda$ (as the parameter 3λ is passed in the `add` call at line 50 of Alg. 19). Similarly, we then subtract H_∞ from $r + 2\lambda$ and propagate the carry all the way to $r + 4\lambda$.

				↗ v'	
r	L_0	$D_0 + H_0 - L_\infty$	$\to D_1 - L_0 - H_\infty$	$\to D_2 - H_0 + L_\infty$	H_∞
	λ	λ	λ	λ	$\mu + \nu - \lambda$

s	D_0	D_2	
	λ	$\lambda + 1$	remaining scratch space

We then only have to propagate v' to $r + 4\lambda$ to finish the computation. The proof that this propagation does not overflow is much more straightforward than for Toom-2. Indeed, this time there is only one carry to propagate, so we can rely on the fact that we know for sure that the final result fits in space $3\lambda + \mu + \nu$. This also proves that if $\mu + \nu = \lambda$ (so there is no space to propagate the carry to), then v' must be 0.

4.4.3 General case

When the two operands have sufficiently similar sizes, we can compute their product using either Toom-2 or Toom-2.5. When the operand sizes are very unbalanced, none of these algorithms can be used directly without violating their preconditions. In this case, WhyMP performs a block product by decomposing the larger operand in blocks of sizes 3/2 times that of the smaller operand, and calling Toom-2.5 repeatedly (Alg. 21). This is a small departure from GMP. GMP implements a third Toom-Cook algorithm called `toom42`, which expects the larger operand to be twice as long as the smaller one. It calls `toom42` repeatedly to perform the general case multiplication, rather than `toom32`. I chose to not verify `toom42` due to time constraints, expecting the performance difference to not be egregious as `toom42` is not called by any of the recursive calls in GMP. More detailed benchmarks can be found in Sec. 4.8.2.

The general case multiplication function is meant to be exposed to the user, with previous multiplication routines left internal. It calls other multiplication algorithms that depend on the operand sizes. If one operand is very small, the schoolbook multiplication is the best one (line 3). Otherwise, if the operands are of sufficiently similar sizes, then we simply call Toom-2 or Toom-2.5 (line 34 of Alg. 21). Finally, if the operands are large and very unbalanced, we need to perform a block product.

The main loop invariants are as follow:

$$\begin{aligned}
\texttt{value}(r, m + n - u) &= \texttt{value}(a, m - u) \cdot \texttt{value}(b, n) \\
a' &= a + m - u \\
r' &= r + m - u.
\end{aligned}$$

At the beginning of a loop iteration, a' points to the first limb of a that has not been multiplied yet, and r' points to the zone in r where the next subresult

Algorithm 21 General case multiplication.

```
Require: valid($r, m + n$), valid($a, m$), valid($b, n$)
Require: $0 < n \leq m$
Ensure: value($r, m + n$) = value($a, m$) $\cdot$ value($b, n$)
function MUL($r, a, m, b, n$)
    if $n < 30$ then                 ▷ Small operands, use schoolbook algorithm.
        MUL_BASECASE($r, a, m, b, n$)
    else
        $k \leftarrow 64$
        $s \leftarrow$ ALLOC($5n + 128$)      ▷ Allocate sufficiently large scratch space.
        if $2m \geq 5n$ then          ▷ Unbalanced operands, use block product.
            $m' \leftarrow \lfloor 3n/2 \rfloor$                  ▷ Block size (rounded down).
            $w \leftarrow$ ALLOC($4n$)  ▷ Allocate workspace to store Toom-2.5 result.
            $u \leftarrow m$                   ▷ Remaining section of $a$ to multiply.
            TOOM32_MUL($r, a, b, s, m', n, k$)
            $u \leftarrow u - m'$
            $a' \leftarrow a + m'$
            $r' \leftarrow r + m'$
            while $u \geq 2n$ do
                TOOM32_MUL($w, a', b, s, m', n, k$)
                $v \leftarrow$ ADD_N($r', r', w, n$)          ▷ Add result to subtotal.
                COPY($r' + n, w + n, m'$)          ▷ Continue the addition.
                INCR($r' + n, v$)                    ▷ Propagate the carry.
                $u \leftarrow u - m'$
                $a' \leftarrow a' + m'$
                $r' \leftarrow r' + m'$
            if $n \leq u$ then                          ▷ Multiply the last block.
                if $4u < 5n$ then
                    TOOM22_MUL($w, a', b, s, u, n, k$)
                else
                    TOOM32_MUL($w, a', b, s, u, n, k$)
            else                                ▷ Operand sizes are reversed
                MUL($w, b, n, a', u$)
            $v \leftarrow$ ADD_N($r', r', w, n$)
            COPY($r' + n, w + n, u$)
            INCR($r' + n, v$)
        else
            if $4m < 5n$ then
                TOOM22_MUL($r, a, b, s, m, n, k$)
            else
                TOOM32_MUL($r, a, b, s, m, n, k$)
```

should be added. Note that the first n limbs after r' already contain a part of the subtotal.

After multiplying (a', m') by (b, n), we need to add the product (stored in w) to r'. This is done in lines 17-19. Instead of performing one addition of length $m' + n$, we take advantage of the fact that only the first n limbs of r' are occupied. We perform an addition of length n on that zone and simply copy over the rest of w to $r' + n$, and then propagate the carry of the addition.

What happens after the loop is similar to one last iteration of the loop. The only change is that the last block of a that is left to multiply has length u, not exactly $3n/2$. Depending on the ratio between n and u, either Toom-2 or Toom-2.5 is called. If $u < n$, the operands are still balanced enough that the recursive call at line 29 will necessarily jump to line 34 and call one of these two functions, or call the schoolbook algorithm.

4.5 Divide-and-conquer square root

The GMP square root algorithm computes the square root s and the remainder r of its operand a such that $a = s^2 + r$ and $0 \leq r \leq 2s$, or equivalently, $s^2 \leq a < (s+1)^2$.

It consists in four functions. From a verification standpoint, the most interesting one was by far the base case algorithm `sqrtrem1` (Sec. 4.5.1), which computes the square root of a limb. It is essentially a fixed-point arithmetic algorithm, which required a good amount of modeling work. This work was paid off when the Gappa tool turned out to be able to discharge almost all the verification conditions automatically. The second function (Sec. 4.5.2) computes the square root of a 2-limb number and is essentially a wrapper around the first. The third function (Sec. 4.5.3) is a general case divide-and-conquer algorithm. It is far from trivial, but it was already formally verified by Bertot et al. using Coq [11]. As a result, I was able to heavily inspire my proof from theirs. Finally, the last function (Sec. 4.5.4) is a wrapper around the general case algorithm that takes care of normalizing.

4.5.1 Square root, $n = 1$: a fixed-point algorithm

GMP's `mpn_sqrtrem1` function computes the square root of a 64-bit integer. Although it manipulates only integers, it is best understood as a fixed-point arithmetic algorithm that implements Newton's method. The source code is relatively short, but very intricate. We verified it semi-automatically using the Gappa tool [26] in addition to the usual SMT solvers. This section is drawn from a previous paper in which we describe our formal proof of this algorithm [71].

The algorithm

Let us start by briefly explaining the algorithm. It relies on many C-specific features, such as the way numbers are represented in memory, type casts and bitwise logical shifts. Therefore, a pseudocode would not make much sense or be more readable than the original source code. So, let us analyze GMP's implementation directly. The source code is reproduced in Fig. 4.3. For the sake of readability, some comments, whitespace, and variable names were modified, and some macros were expanded.

```
 1  #define MAGIC 0x10000000000
 2
 3  mp_limb_t mpn_sqrtrem1(mp_ptr rp, mp_limb_t a0) {
 4    mp_limb_t a1, x0, x1, x2, c, t, t1, t2, s;
 5    unsigned abits = a0 >> (64 - 1 - 8);
 6    x0 = 0x100 | invsqrttab[abits - 0x80];
 7    // x0 is the 1st approximation of 1/sqrt(a0)
 8    a1 = a0 >> (64 - 1 - 32);
 9    t1 = (mp_limb_signed_t) (0x2000000000000 - 0x30000 - a1 * x0 * x0) >> 16;
10    x1 = (x0 << 16) + ((mp_limb_signed_t) (x0 * t1) >> (16+2));
11    // x1 is the 2nd approximation of 1/sqrt(a0)
12    t2 = x1 * (a0 >> (32-8));
13    t = t2 >> 25;
14    t = ((mp_limb_signed_t) ((a0 << 14) - t * t - MAGIC) >> (32-8));
15    x2 = t2 + ((mp_limb_signed_t) (x1 * t) >> 15);
16    c = x2 >> 32;
17    // c is a full limb approximation of sqrt(a0)
18    s = c * c;
19    if (s + 2*c <= a0 - 1) {
20        s += 2*c + 1;
21        c++;
22      }
23    *rp = a0 - s;
24    return c;
25  }
```

Figure 4.3: Square root of a 64-bit integer.

The function takes an unsigned 64-bit integer `a0` larger than or equal to 2^{62} and returns its square root, storing the remainder in `*rp`. The best way to understand this algorithm is to view it as a fixed-point arithmetic algorithm, with the binary point initially placed such that the input `a0` represents $a \in [0.25; 1]$.

The main part of the algorithm consists in performing two iterations of Newton's method to approximate $a^{-1/2}$ with 32 bits of precision. As part of the last iteration, the approximation is multiplied by a to obtain a suitable approximation of $\sqrt{a}$. More precisely, we are looking for a root of $f(x) = x^{-2} - a$. Given $x_i = a^{-1/2}(1 + \varepsilon_i)$, we define $x_{i+1} = x_i - f(x_i)/f'(x_i) = x_i(3 - ax_i^2)/2$. Furthermore, if we pose ε_{i+1} such that $x_{i+1} = a^{-1/2}(1 + \varepsilon_{i+1})$, we find $|\varepsilon_{i+1}| \approx \frac{3}{2} \cdot |\varepsilon_i|^2$.

Note that the iteration can be computed with only additions, multiplications, and logical shifts, which all take considerably fewer cycles than a division. For instance, we compute x_1 as $x_0 + x_0 t_1/2$, with $t_1 \approx 1 - ax_0^2$ at line 9. The division by 2 is implicitly performed by the right shift at line 10. The absence of division primitives is the main reason why the algorithm looks for an approximation of the inverse square root rather than the square root itself, which would involve a division by x_i at each step of the iteration.

The initial approximation of $a^{-1/2}$ is taken from the precomputed array `invsqrttab` of 384 constants of type `char`. Using interval arithmetic, we have checked exhaustively that the initial approximation `x0` has a relative error ε_0 smaller than about $2^{-8.5}$ for all values of `a0`. It follows that after an iteration, we have $|\varepsilon_1| \lesssim 2^{-16.5}$ and after two steps, $|\varepsilon_2| \lesssim 2^{-32.5}$. The square root of `a0` can be represented using at most 32 bits, so we would expect the final approximation to be either exactly the truncated square root of a0 or off by one. Note that we are computing using fixed-point numbers rather than real numbers, so additional rounding errors are introduced during the computation and worsen this error bound somewhat. However, the quadratic nature of the convergence absorbs most of the rounding errors from the first iteration, and the final result is still off by at most one. The magic constants 0x30000 and MAGIC, which would not be part of an ideal Newton iteration, ensure that the approximation is always by default. As a result, the approximation after two steps is known to be either exactly the truncated square root, or its immediate predecessor. The final if block performs the required adjustment step, by adding 1 to the result if it does not make its square exceed a0.

There are several implementation tricks that make this algorithm both efficient and difficult to prove. For instance, some computations intentionally overflow, such as the left shift and the multiplication at line 14. In this case, t is represented using 39 bits, and a0 uses all 64 bits, so both computations overflow by 14 bits. However, the top part of t*t is known to be equal to the top part of a0, and these numbers are subtracted, so no information is lost. Another thing to note is that the variable t, which is an error term, essentially represents a signed value even though it is an unsigned 64-bit integer. Indeed, we cast it into a signed integer before shifting it to the right. This means that its sign is preserved by the shift. However, t could not be represented as an actual signed integer because some computations involving it overflow, such as the product t*t at line 14.

Modeling fixed-point arithmetic inside Why3

For the most part, the C code uses only one number type: unsigned 64-bit integers. Relevant arithmetic operations are addition, multiplication, left shift, and right shift. We could write the WhyML code using our existing model of unsigned integers, but we would lose some helpful information about the algorithm. Indeed, the integers that the algorithm manipulates are fixed-point representations of real numbers. So, it is important to carry around the position of their binary point. To do so, an initial idea could be to introduce a new WhyML record type with two fields:

```
type fxp = { ival: uint64; ghost iexp: int }
```

The iexp field denotes the position of the binary point. As it is a ghost field, from an implementation point of view, a value of type fxp is indistinguishable from the value of its field ival, which is an unsigned 64-bit integer. The extraction mechanism features a record optimization mechanism that simplifies records with a single non-ghost field into values of the type of that field. So, after extraction, values of type fxp would be extracted to 64-bit unsigned integers, which is exactly what we want.

We also need a way to relate fixed-point integers to the real numbers they represent. WhyML features an axiomatized type real that represents real numbers. Much like the int type of unbounded integers, it is meant to be used in specifications, lemmas and proofs, but not in extracted programs. Let us for now assume that we have a function rval that maps fxp values to real numbers, and leave its definition for later.

```
function rval (x:fxp): real
```

We can now specify the basic arithmetic operations on fixed-point values. As an illustration, here is part of the declaration of addition. It takes two fixed-point numbers x and y as inputs and returns a fixed-point number denoted *result* in the specification. The precondition requires that the binary points of x and y are aligned; the user will have to prove this property. The postcondition ensures that the result is aligned with x (and thus y too); this property will be available in any subsequent proof.

```
val fxp_add (x y: fxp): fxp
  requires { iexp x = iexp y }
  ensures  { iexp result = iexp x }
  ensures  { rval result = rval x +. rval y }
```

The last line of the specification ensures that the real number represented by *result* is the sum of the two real numbers represented by x and y. The "`+.`" operator is the addition on reals. Of course, in the extraction driver, we specify that `fxp_add` should simply be replaced by the `(+)` operator. However, this is when we run into trouble.

Let us come back to the issue of defining `rval`. An initial idea would be to define it intuitively as $rval = ival \cdot 2^{iexp}$. Unfortunately, that would make the algorithm impossible to prove. Indeed, with this definition of `rval`, the above specification of `fxp_add` would be an incorrect model of C integer addition due to wraparound. A fix would be to add a `requires` clause that forbids the sum `ival x + ival y` to overflow. However, the algorithm that we want to prove violates this precondition! Indeed, as mentioned before, the algorithm features intentional overflows. When they occur, the real numbers that are represented may indeed exceed 2^{64}, which the proposed definition of `rval` disallows. So, we need a more subtle specification for `fxp` operations.

To circumvent this issue, we add `rval` as another ghost field of `fxp`. We also add two type invariants to state that the values of the three fields are related. Now, whenever the code creates a fixed-point value, the user has to prove that the invariants hold.

```
type fxp = { ival: uint64; ghost rval: real; ghost iexp: int }
  invariant { rval = floor_at rval iexp }
  invariant { ival =
    mod (floor (rval *. pow2 (-iexp))) (uint64'maxInt + 1) }
```

The first invariant forces the real number to be a multiple of 2^{iexp} by stating that *rval* is left unchanged by truncating it at this position. The second invariant states that the 64-bit integer can be obtained by first scaling the real number so that it becomes an integer, and then making it fit into the `uint64` type using the remainder of an Euclidean division. The `floor_at` function that appears in the first invariant is defined as follows:

```
function floor_at (x: real) (k: int): real =
  floor (x *. pow2 (-k)) *. pow2 k
```

A fixed-point value can be created using the following function. It takes a 64-bit integer, usually a literal one, and the position of the binary point. Since this position is ghost, and so are the fields `iexp` and `rval`, this function is effectively the identity function.

```
let fxp_init (x: uint64) (ghost k: int): fxp
= { ival = x; iexp = k; rval = x *. pow2 k }
```

Subtraction and multiplication are not fundamentally different from addition. Left and right shifts are more interesting. Contrarily to plain integer arithmetic, their role is not just to perform some cheap multiplication or division by a power of two; they can also be used to move the binary point to a given position. Let us illustrate this situation with the following function, which performs an arithmetic right shift "(mp_limb_signed_t)x >> k".

```
val fxp_asr' (x: fxp) (k: uint64) (ghost m: int): fxp
  requires { int64'minInt *. pow2 (iexp x) ≤.
    rval x ≤. int64'maxInt *. pow2 (iexp x) }
  ensures { iexp result = iexp x + k - m }
  ensures { rval result =
    floor_at (rval x *. pow2 (-m)) (iexp x + k - m) }
```

The precondition requires that the real number represented by x is bounded, so that the sign bit contains enough information to fill the $k+1$ most significant bits. In particular, to use this function, the user will have to prove that, if some previous operation overflowed, it has been compensated in some way. The ghost argument m tells how much of the shift is actually a division of the real number, so the binary point is moved by $k-m$ as stated by the first postcondition. (The `fxp_asr` variant corresponds to $m = 0$.) Finally, the second postcondition expresses that the real result is $x \cdot 2^{-m}$ except for the least significant bits that are lost. One example of this occurs at line 10 of the square root. The quantity `x0 * t1` is right-shifted by 18 bits, but this should be interpreted as a division by 2 that also truncates the 17 least significant bits. In the WhyML transcription, the right shift is translated as a call to `fxp_asr'` with $k = 18$ and $m = 1$.

This loss of accuracy is expressed using the aforementioned `floor_at` function. This function can be interpreted as a rounding toward $-\infty$ in some fixed-point format. As such, when verification conditions will be sent by Why3 to Gappa, an expression "`floor_at x p`" will be translated to "`fixed<p,dn>(x)`". Note that Gappa requires `p` to be an integer literal, so it would choke on "`iexp x + k - m`". So, we improved Why3 a bit to make it precompute such expressions before sending them to Gappa. As a side effect, this change also made it possible to turn constant expressions such as "`int64'minInt *. pow2 (iexp x)`" into proper numbers. That is the only change we had to make to Why3.

WhyML transcription and proof

Once fixed-point arithmetic has been modeled as a Why3 theory, translating GMP's square root algorithm from C code to WhyML is mostly straightforward. The only details we had to guess are the ghost arguments passed to functions `fxp_init` and `fxp_asr'`, which are obvious only if one knows that the algorithm implements Newton's method. The complete specification and part of the WhyML code are given in Figure 4.4. The specification requires that the function is called with a valid pointer `rp` and an integer `a0` large enough. It ensures that the unsigned integer returned by the function is the truncated square root and that the unsigned integer pointed by `rp` holds the remainder.

The main difference between this WhyML transcription and the original C code is the call to `rsa_estimate` at line 11. In the original C code, the initial estimate of the inverse square root of `a0` is looked up from a global array. Sadly, Why3 does not yet support this, so we abstract it in a `val` declaration. The

```
let sqrt1 (rp: ptr uint64) (a0: uint64): uint64
  requires { valid rp 1 }
  requires { 0x4000000000000000 ≤ a0 }
  ensures { result*result ≤ a0 < (result+1)*(result+1) }
  ensures { result*result + get rp = a0 }
=
  let a = fxp_init a0 (-64) in
  assert { 0.25 ≤. a ≤. 0xffffffffffffffffp-64};
  assert { 0. <. a };
  let ghost rsa = 1. /. sqrt a in
  let x0 = rsa_estimate a in
  let ghost e0 = (x0 -. rsa) /. rsa in
  let a1 = fxp_lsr a 31 in
  let ghost ea1 = (a1 -. a) /. a in
  let m1 = fxp_sub
    (fxp_init 0x2000000000000 (-49))
    (fxp_init 0x30000 (-49)) in
  let t1' = fxp_sub m1 (fxp_mul (fxp_mul x0 x0) a1) in
  let t1 = fxp_asr t1' 16 in
  let x1 = fxp_add (fxp_lsl x0 16) (fxp_asr' (fxp_mul x0 t1) 18 1) in
  let ghost mx1 = x0 +. x0 *. t1' *. 0.5 in
  assert { (mx1 -. rsa) /. rsa =
    -0.5 *. (e0*.e0 *. (3.+.e0) +. (1.+.e0) *.
      (1.-.m1 +. (1.+.e0)*.(1.+.e0) *. ea1)) };
  ...
```

Figure 4.4: WhyML square root up to the assertion on the first Newton iteration.

specification of `rsa_estimate` is reproduced below. Its postcondition specifies an upper bound of the relative error on the initial estimate. In the extraction driver, we reproduce the `invsqrttab` array from the original GMP code and replace calls to `rsa_estimate` by lookups in the array. This means that the extracted code ends up very similar to the original code. However, the error bound in the postcondition is an axiom in our Why3 development. Since the array contains only a few hundred elements, we were able to simply verify it outside Why3.

```
val rsa_estimate (a: fxp): fxp
  requires { 0.25 ≤. a ≤. 0xffffffffffffffffp-64 }
  requires { iexp a = - 64 }
  ensures { iexp result = -8 }
  ensures { 256 ≤ ival result ≤ 511 }
  ensures { 1. ≤. rval result ≤. 2. }
  ensures { let rsa = 1. /. sqrt a in
            let e0 = (rval result -. rsa) /. rsa in -0.00281 ≤. e0 ≤. 0.002655
    }
```

Let us now describe the proof of the algorithm proper. The most critical part of the algorithm is the verification of the following assertion, which states that the fixed-point value x approximates $sa = 2^{-32}\lfloor\sqrt{a_0}\rfloor$ with an accuracy of 32 bits:

```
assert { -0x1.p-32 ≤. x -. sa ≤. 0. };
```

This is exactly the kind of verification condition Gappa is meant to verify. Unfortunately, Gappa is unable to automatically discharge it, as it does not know anything about Newton's method. So, we have to write additional assertions in the code to help Gappa. In particular, we have to make it clear that the convergence is quadratic. For example, the following equality gives the relative error between $a^{-1/2}$ and an idealized version x'_1 of the second approximation `x1`.

$$\frac{x'_1 - a^{-1/2}}{a^{-1/2}} = -\tfrac{1}{2}(\varepsilon_0^2(3+\varepsilon_0)+\ldots).$$

The value x'_1, denoted by `mx1` in Figure 4.4, is not exactly the theoretical x_1 from the Newton iteration, as it replaces a by `a1` and takes into account the magic constant `0x30000`. However, the formula does not contain any rounding operator, so it can be obtained using a computer algebra system without much trouble. From this formula, Gappa will then deduce a way to bound the relative error between $a^{-1/2}$ and `x1`, despite the rounding errors.

All in all, Gappa needs four equalities to be able to prove the main error bound. Two of them are the relative errors of both Newton iterations. The other two are just there to help Gappa fight the dependency effect of interval arithmetic, e.g., $a \cdot a^{-1/2} = \sqrt{a}$. All these equalities are written as WhyML assertions and have to be discharged by Why3 using some other prover. Fortunately, the `field` tactic makes it straightforward to do so using the Coq proof assistant, once it has been told $a = \sqrt{a}^2$. In the end, the most difficult part of the proof was to determine the contents of these assertions, which was not that hard.

The verification went surprisingly well and Gappa was able to discharge all the preconditions of the right-shift functions. It failed at discharging the main error bound, though. (It would have presumably succeeded, had we let it run long enough.) We first tried to modify slightly the equality describing the relative error of the second Newton iteration. This made the verification faster, but it would still have taken hours at best. So, we ended up modifying the code a bit. GMP's square root makes use of a second magic constant 2^{40} (line 14 of Fig. 4.3); we replaced it by $2.25 \cdot 2^{40}$. Gappa was then able to discharge the verification condition in a few seconds.

This does not mean that GMP's algorithm is incorrect. It just means that Gappa overestimates some error, which prevents it from proving the code for the original constant. Interestingly enough, a comment in the C code indicates that this magic constant can be chosen between $0.9997 \cdot 2^{40}$ and $35.16 \cdot 2^{40}$.

Once we have proved $x - sa \in [-2^{-32}; 0]$, proving $c - \lfloor\sqrt{a}\rfloor \in [-1; 0]$ (with c the integer representing x) is mostly a matter of unfolding the definitions and performing some simplifications. Then, by stating a suitable assertion, we get SMT solvers to automatically prove that none of the operations in the computation of $c \cdot c + 2c$ can overflow. All that is left is verifying the postcondition of the function, which SMT solvers have no difficulty discharging automatically.

The verification of this function ended up very different from the other proofs in this chapter. The reason is that the most difficult part of the proof (the assertion that the estimate after two Newton steps has 32 bits of precision) was actually performed automatically. For the other algorithms, the non-trivial parts of the proof are performed using long Why3 assertions. Of course, these assertions are checked by automated solvers, but they are long precisely because many intermediary steps are needed to make the solvers accept them. From the point of view of the programmer, writing these long assertions is not so different from writing a pen-and-paper proof, in that a similar level of detail and understanding is required and it takes a similar amount of time. For this square root algorithm, it was not so. Indeed, I would not have been able to write a rigorous pen-and-paper proof that the 32-bit estimate is accurate, or even understand the algorithm well enough to be convinced that it is correct. Yet, I managed to get Why3 and Gappa to prove the correctness of the algorithm. The main reason for this success is that the hard part of the proof is exactly the sort of problem that Gappa is designed to handle. Furthermore, it was easy to separate this core problem from the subproblems that Gappa cannot handle well (the equalities on the relative errors) and treat these separately. Conversely, the goals that require large assertions do not quite fit in the domain of any of the automated solvers, so more user effort is required. Chapter 5 further discusses this effort, as well as an attempt to increase the degree of automation of these proofs.

4.5.2 Square root, $n = 2$

The `sqrtrem2` function (Alg. 22) computes the square root of a, which must be two limbs long. It takes two destinations operands, s to store the square root of a and r to store the remainder. It returns the high limb c of the remainder, such that $\texttt{value}(a, 2) = s[0]^2 + \beta c + r[0]$. The parameters r and s are pointers that only need space for 1 limb. We assume a to be normalized such that its high limb is greater than or equal to $\beta/4$. The algorithm computes an initial estimate of the square root by calling `sqrtrem1` on the high limb of a, and then adjusts toward the correct square root.

Initial estimate (lines 2-4)

After the call to `sqrtrem1` at line 2, we have $s_0^2 + r[0] = a_1$ and $a = 2^{64}a_1 + a_0$. The value $2^{32}s_0$ is the initial estimate for the square root of a. We compute an initial remainder r_0 such that $2^{33}r_0 + (a[0] \bmod 2^{33}) = \beta r[0] + a[0]$.

First adjustment (lines 5-13)

We compute the quotient q of r_0 by s_0, such that at line 6, $s_0 q$ is somewhat close to r_0 and $q < 2^{32}$. We pose $s_1 = 2^{32}s_0 + q$ as a candidate square root. Indeed, $2^{33}qs_0 \approx \beta r[0] + a[0]$, so at line 8: $s_1^2 = \beta s_0^2 + 2^{33}qs_0 + q^2 \approx \beta a[1] + a[0] + q^2$. More precisely,

$$s_1^2 = \beta a[1] + a[0] + q^2 - (2^{33}(r_0 - qs_0) + a[0] \bmod 2^{33}).$$

We pose $u = r_0 - qs_0$, and r_1, c such that $r_1 + \beta c = 2^{33}(r_0 - qs_0) + a[0] \bmod 2^{33}$, $r_1 \in [0, \beta - 1]$. Therefore, at line 11 we have $s_1^2 + r_1 + \beta c = a + q^2$. At lines 12-13, we subtract q^2 from $r_1 + \beta c$ in a way that avoids arithmetic underflows. At that point, $s_1^2 + r_1 + \beta c = a$.

No underapproximation of the square root

Let us show that s_1 is not an underapproximation of the square root, or in other terms, $\lfloor\sqrt{a}\rfloor \leq s_1$. This is equivalent to $a < (s_1 + 1)^2$, or $r_1 + \beta c \leq 2s_1$. We have $r_1 + \beta c = 2^{33}u + a_l - q^2$.

If $q_0 < 2^{32}$, then $q = q_0$ and $u = r_0 \bmod s_0 < s_0$, and $a_l < 2^{33}$, so $r_1 + \beta c \leq 2^{33}s_0 \leq 2s_1$.

The only remaining case is $q_0 = 2^{32}, q = 2^{32} - 1$. In this case, we notice that $r[0] \leq 2s_0$ (postcondition of `sqrtrem1`). This implies $r_0 \leq 2^{32}s_0 + a_h$, so $u = r_0 - (2^{32} - 1)s_0 \leq s_0 + a_h$. We also have $\beta = (q+1)^2$, so:

$$\begin{aligned}
2^{33}u + a_l - q^2 &\leq 2^{33}s_0 + a[0] - q^2 \\
&\leq 2^{33}s_0 + \beta - 1 - q^2 \\
&= 2^{33}s_0 + 2q \\
&\leq 2s_1.
\end{aligned}$$

Avoiding overapproximation (lines 14-19)

Let us now show that $s_1 < \lfloor\sqrt{a}\rfloor + 1$, or equivalently, $s_1^2 \leq a$ (no overapproximation of the square root). If we had $s_1^2 > a$, it would imply $r_1 + \beta c < 0$, or equivalently, $c \leq -1$. The conditional at line 14 takes care of this case by adding $2s_1 - 1$ to $r_1 + \beta c$ and removing 1 from s_1, leaving the sum $s_1^2 + r_1 + \beta c$ unchanged and avoiding arithmetic overflows.

Algorithm 22 Square root of a 2-limb number.

```
Require: valid(a, 2), valid(s, 1), valid(r, 1)
Require: $a[1] \geq \beta/4$
Ensure: value(a, 2) = $s[0]^2 + \beta \cdot$ result $+ r[0]$
Ensure: $0 \leq$ result $\leq 1$
Ensure: $r[0] + \beta \cdot$ result $\leq 2s[0]$
 1: function SQRTREM2(s, r, a)
 2:     $s_0 \leftarrow$ SQRTREM1(r, a[1])
 3:     $a_h \leftarrow a[0] \gg 33$
 4:     $r_0 \leftarrow (r[0] \ll 31) + a_h$
 5:     $q_0 \leftarrow \lfloor r_0/s_0 \rfloor$                          ▷ $q_0 \leq 2^{32}$.
 6:     $q \leftarrow q_0 - (q_0 \gg 32)$              ▷ If $q_0 = 2^{32}$, reduce it by 1.
 7:     $u \leftarrow r_0 - qs_0$
 8:     $s_1 \leftarrow (s_0 \ll 32) + q$
 9:     $c \leftarrow u \gg 31$
10:     $a_l \leftarrow a[0] \bmod 2^{33}$
11:     $r_1 \leftarrow (u \ll 33) + a_l$
12:     $c \leftarrow c - (r_1 < q^2)$                             ▷ $-1 \leq c$.
13:     $r_1 \leftarrow r_1 - q^2 \bmod \beta$
14:     if $c < 0$ then                      ▷ Square root too large, adjust.
15:         $r_1 \leftarrow r_1 + s_1 \bmod \beta$
16:         $c \leftarrow c + (r_1 < s_1)$                   ▷ Carry propagation.
17:         $s_1 \leftarrow s_1 - 1$
18:         $r_1 \leftarrow r_1 + s_1 \bmod \beta$
19:         $c \leftarrow c + (r_1 < s_1)$                   ▷ Carry propagation.
20:     $r[0] \leftarrow r_1$
21:     $s[0] \leftarrow s_1$
22:     return $c$
```

4.5.3 Square root, general case

My Why3 proof of the general case divide-and-conquer square root algorithm is largely lifted from Bertot et al. [11]. Their article describes their Coq formalization and provides a detailed paper proof of a transcription of the algorithm in pseudocode. If I described my Why3 proof here in the same style as the other algorithms, it would end up extremely similar to their proof, with only tiny adjustments to account for small changes in the GMP implementation since the publication of their article. Therefore, only the signature and specification of the algorithm are given in Alg. 23. The variable w is a scratch buffer meant to store intermediate results.

Algorithm 23 Square root of a normalized integer (specification).

```
Require: $\texttt{valid}(a, 2n)$, $\texttt{valid}(s, n)$, $\texttt{valid}(w, \lfloor n/2 \rfloor + 1)$
Require: $1 \leq n$
Require: $a[2n-1] \geq \beta/4$
Ensure: $\texttt{value}(s,n)^2 + \texttt{value}(a,n) + \beta^n \texttt{result} = \texttt{value}(\texttt{old}\ a, 2n)$
Ensure: $\texttt{value}(a,n) + \beta^n \texttt{result} \leq 2 \cdot \texttt{value}(s,n)$
Ensure: $0 \leq \texttt{result} \leq 1$
  function DC_SQRTREM$(s, a, w, n)$
```

4.5.4 Square root, normalizing wrapper

The previous "general case" algorithm still requires its input to have even length and to be normalized (the highest limb of a must be greater than or equal to $\beta/4$). The final algorithm (Alg. 24) is essentially a wrapper around the two previous ones that normalizes its operand, calls the appropriate square root function, and denormalizes the result. It returns the size of the remainder in limbs. If the result is 0, the operand is a perfect square. The key idea of this algorithm is the following lemma from Bertot et al., which justifies that simply denormalizing the square root of the normalized operand gives the correct square root [11].

Lemma 8 (normalization)**.** *Let N, N_1, S_1, c such that $S_1^2 \leq N_1 < (S_1+1)^2$, $N_1 = 2^{2c}N$. Let S, s_0 such that $S_1 = 2^c S + s_0$, $0 \leq s_0 < 2^c$. Then $S^2 \leq N < (S+1)^2$.*

We first compute c, the floor of half the number of leading zeros of the high limb of a. This means that shifting a to the left by $2c$ will multiply it by a power of 2 such that at most one leading zero remains, which is exactly the precondition of the previous square root algorithms.

Special case $n = 1$

This case is a direct application of the lemma above. If the operand is not already normalized, we shift it to the left by $2c$ and shift the result to the right by c. The lemma ensures that this yields the correct square root, and we compute a remainder straightforwardly.

Algorithm 24 Square root of an integer.

```
Require: $\texttt{valid}(s, \lfloor n/2 \rfloor + 1), \texttt{valid}(r, n), \texttt{valid}(a, n)$
Require: $1 \leq n$
Require: $a[n-1] > 0$
Ensure: $\texttt{value}(a, n) = \texttt{value}(s, \lfloor n/2 \rfloor + 1)^2 + \texttt{value}(r, \texttt{result})$
Ensure: $\texttt{value}(r, \texttt{result}) \leq 2 \cdot \texttt{value}(s, n)$
Ensure: $\texttt{result} > 0 \implies r[\texttt{result} - 1] > 0$
function SQRTREM$(s, r, a, n)$
    $h \leftarrow a[n-1]$
    $c = $ COUNT_LEADING_ZEROS$(h)/2$
    if $n = 1$ then
        if $c = 0$ then
            $s[0] = $ SQRTREM1$(r, h)$
        else
            $h' \leftarrow h \ll 2c$
            $s' \leftarrow $ SQRTREM1$(r, h') \gg c$
            $s[0] \leftarrow s'$
            $r[0] \leftarrow h - s'^2$
        if $r[0] = 0$ then
            return 0
        else
            return 1
    $k \leftarrow (n+1)/2$
    $w \leftarrow $ ALLOC$(\lfloor k/2 \rfloor + 1)$
    $t \leftarrow $ ALLOC$(2k)$
    if $n = \mod 2 = 1 \vee c \neq 0$ then
        $t[0] \leftarrow 0$
        $t' \leftarrow t + (n \mod 2)$
        if $c \neq 0$ then
            LSHIFT$(t', a, n, 2c)$
        else
            COPY$(t', a, n)$
        if $n \mod 2 = 1$ then
            $c \leftarrow c + 32$
        $r_l \leftarrow $ DC_SQRTREM$(s, t, w, k)$
        $s_0 \leftarrow $ ALLOC$(1)$
        $s_0[0] \leftarrow s[0] \mod 2^c$                                ▷ $c \leq 63$.
        $r_l \leftarrow r_l + $ ADDMUL_1$(t, s, k, 2s_0[0])$
        $b \leftarrow $ SUBMUL_1$(t, s_0, s_0[0], 1)$
        if $k > 1$ then
            $b \leftarrow $ SUB_1$(t+1, t+1, b, k-1)$
        $r_l \leftarrow r_l - b$
        RSHIFT$(s, s, k, c)$
        $t[k] \leftarrow r_l$
```

```
38:         $c_2 \leftarrow 2c$
39:         if $c_2 < 64$ then
40:             $k \leftarrow k + 1$
41:         else
42:             $t \leftarrow t + 1$
43:             $c_2 \leftarrow c_2 - 64$
44:         if $c_2 \neq 0$ then
45:             RSHIFT$(r, t, k, c_2)$
46:         else
47:             COPY$(r, t, k)$
48:         $r_n \leftarrow k$
49:     else
50:         COPY$(r, a, n)$
51:         $h \leftarrow$ DC_SQRTREM$(s, r, w, k)$                    ▷ $0 \leq h \leq 1$
52:         $r[k] \leftarrow h$
53:         $r_n \leftarrow k + h$
54:     NORMALIZE$(r, r_n)$
55:     return $r_n$
```

General case

We compute $k = \lfloor (n+1)/2 \rfloor$ such that $2k = n + (n \bmod 2)$. If the conditional at line 19 is true, then a is either not normalized or of odd length, so we must normalize it before calling `dc_sqrtrem`. After line 27, this is done and we have $\texttt{value}(t, 2k) = 2^{2c}\texttt{value}(a, n)$. The value of c is incremented by 32 if n is odd, so that we shift a by an extra limb (which is what the computation of t' does) and end up with an even-length number.

At that point, we can call `dc_sqrtrem`. The normalization lemma implies that $\lfloor \frac{\texttt{value}(s,k)}{2^c} \rfloor$ is the correct square root (using the same variables as the lemma, $S = \lfloor S_1 2^{-c} \rfloor$). We still need to compute the remainder.

At line 29, we pose $S_1 = \texttt{value}(s, k)$, $R_1 = \texttt{value}(t, k) + \beta^k r_l$, $N = \texttt{value}(a, n)$, $N_1 = 2^{2c}N$. We have $S_1^2 + R_1 = N_1$. We pose $s_0 = S_1 \bmod 2^c$, so that N_1 can be written as $(S_1 - s_0)^2 + 2S_1 s_0 - s_0^2 + R_1$. We add $2S_1 s_0$ to t at line 31, and then subtract s_0^2 at line 32. After line 32, $\texttt{value}(t, k) + \beta^k r_l - \beta b = R_1 + 2S_1 s_0 - s_0^2$. Note that the borrow b has not yet been propagated, instead the subtraction was only over a length of 1. The propagation occurs at lines 33-34, and after line 35 we have $\texttt{value}(t, k) + \beta^k r_l = R_1 + 2S_1 s_0 - s_0^2$. Since the latter quantity is non-negative, the subtraction at line 35 cannot overflow. At line 37 we write r_l at $t[k]$ such that $\texttt{value}(t, k+1) = R_1 + 2S_1 s_0 - s_0^2$.

We pose S such that $S_1 = s_0 + 2^c S$. The normalization lemma implies that S is the square root of N. At line 36, we denormalize s such that $\texttt{value}(s, k) = S$. Furthermore, $S_1 - s_0 = 2^c S$, so t holds the remainder of the square root: $\texttt{value}(t, k+1) = R_1 + 2S_1 s_0 - s_0^2 = N_1 - 2^{2c}S^2 = 2^{2c}(N - S^2)$. The only remaining thing to do is to shift t by $2c$. As c can be up to 63, $2c$ can be up to 126 and `rshift` only accepts parameters smaller than 64. The conditional at lines 39-43 takes care of this. If $c_2 \geq 64$, then the low limb of t is all zeroes, so we can cheaply shift t by 64 by simply incrementing the pointer t. At line 44, $\texttt{value}(t, k) = 2^{c_2}(N - S^2)$ and $0 \leq c_2 \leq 63$, so we only have to shift t to the right to get the correct remainder.

Normalized case

If the operand is already normalized and of even length, we can simply call dc_sqrtrem. This is done at lines 50-53.

Normalizing the remainder

At line 54, r_n contains an upper bound on the length of the remainder. However, any number of high limbs may be zero. We call normalize, a small helper function that loops through the higher limb and decreases r_n for each zero limb until a non-zero limb is found or $r_n = 0$.

4.6 The mpz layer

The mpz layer of GMP supports relative integers. It is essentially a wrapper around mpn that keeps track of number signs and sizes. Most mpz functions perform almost no arithmetic except calling the appropriate mpn functions. A relative integer is represented in the mpz layer by a pointer to a record with three fields: an mpn pointer, the size of the number in limbs, and the length of the allocated block. The size is a relative integer and also encodes the sign of the number. The definition of the mpz_ptr type from the GMP source code is reproduced below.

```
typedef struct
{
  int _mp_alloc; /* Number of *limbs* allocated and pointed
                         to by the _mp_d field. */
  int _mp_size; /* abs(_mp_size) is the number of limbs the
                         last field points to. If _mp_size is
                         negative this is a negative number. */
  mp_limb_t *_mp_d; /* Pointer to the limbs. */
} __mpz_struct;

typedef __mpz_struct *mpz_ptr;

#define SIZ(x) ((x)->_mp_size)
#define ABSIZ(x) ABS (SIZ (x))
#define PTR(x) ((x)->_mp_d)
#define ALLOC(x) ((x)->_mp_alloc)
```

The main challenge to the verification of mpz functions is aliasing. Let us take the multiplication function as an example. The function mpz_mul takes three parameters of type mpz_ptr. For example, mpz_mul(w, u, v) computes the product of u and v and stores the result in w. To compute the product, it simply needs to call the mpn multiplication function. However, its code is not trivial. It has to handle a number of cases depending on whether w is equal to either u or v. So, the parameters of the mpz function may or may not be aliased. When this situation occurred in mpn functions, we typically verified several variants of the function (one for the case of aliased parameters, and one for the case of separate parameters). We applied the trick from Sec. 3.3.4 to reduce code duplication when relevant. We cannot do the same for mpz functions without trivializing them entirely. In some cases, handling aliasing issues is all they do apart from calling mpn functions! So, we need to address the issue head on, and actually verify WhyML functions whose parameters may or may not be aliased. This means that we cannot simply model the mpz_struct type as a record that contains an mpn pointer from the usual memory model. Indeed, we need to avoid putting Why3 regions in the mpz_ptr objects, and track aliasing in some other way.

WhyMP currently implements about twenty-five mpz functions. Most of them are relatively trivial wrappers around mpn functions. There are also many repeats. For example, there are five comparison functions, which handle various cases: comparison of a large integer and a machine signed integer, machine unsigned integer, another large integer, comparison of absolute values, and so on. It would not be interesting to go into the details of each function. Let us first present our memory model of mpz (Sec. 4.6.1), and then go over a few functions which posed interesting verification challenges: comparison (Sec. 4.6.2), addition (Sec. 4.6.3), and division (Sec. 4.6.4).

4.6.1 Model

I have developed an ad-hoc WhyML model of mpz objects. The design constraints are as follows. First, there should be an mpz_ptr type that maps to GMP's definition after extraction. Second, it should not contain any region. Third, we should be able to perform all the operations that are expected from mpz_ptr objects. In particular, we need to be able to get from any valid mpz_ptr a valid pointer to limbs that store the expected values.

Since we cannot store aliasing information in the mpz_ptr objects themselves, we model the memory as some global store. We model mpz_ptr objects as an abstract type with an equality operation (extracted to the == equality operator).

```
type mpz_ptr

val predicate mpz_eq (x y: mpz_ptr)
  ensures { result ↔ x = y }
```

The mpz store

The actual contents of all mpz objects are stored in a single global ghost store. Its definition is in Fig. 4.5. From an mpz_ptr object, one should be able to get the following information: a pointer to its limbs, the number of limbs, and the length of the allocated zone. We define the type mpz_memo of a global store with six different fields, and declare a global variable mpz of that type. All mpz functions write in this global variable.

The mpz_memo type cannot store pointers to limbs directly with a map such as mpz_ptr -> ptr limb, for the same reason we cannot define Why3 types that carry an unbounded amount of regions. So, we need to store the limb information in a roundabout way. The abs_value_of field stores the absolute value of each mpz number as a mathematical integer. The sign is stored in sgn, and the number of limbs in abs_size. The alloc field stores the length of the allocated pointers. The logical function value_of maps mpz pointers to the integer they represent.

It would also have been possible to define a size field containing the product of sgn and abs_size, equivalent to the actual contents of the _mp_size field in the actual C struct. Separating this information into two separate fields happens to be more convenient for the proofs.

Using these fields, it is easy to define getters that are equivalent to (and extracted to) the SIZ, ABSIZ and ALLOC macros. They are the functions size_of, abs_size_of and alloc_of respectively. The difficult part is to model the PTR macro. We need some function that takes a mpz_ptr and returns a pointer to limbs. The alloc, abs_size and abs_value_of fields contain all the relevant information on the length and contents of the pointer. However, there are consistency and data race issues. If we get a pointer from the PTR macro and update the pointed block, we need to update the contents of the mpz structure. If PTR is used twice on the same mpz_ptr object, the two resulting pointers also need to be aliased.

One solution to the data race issue is to restrict calls to the PTR macro so that no data races can occur. This is where the readers field comes into play. Calls to the PTR macro can be seen as borrowing the ownership of the mpz_ptr in question from the global store. We distinguish between read-only borrows and read-write ones. The global store contains an accurate representation of the contents of a pointer only as long as no read-write borrow is ongoing. We allow either any number of simultaneous read-only calls to PTR, or a single read-write one, but not both at the same time. This is analogous to the rules that govern borrowing in the Rust programming language. The number of ongoing borrows is represented in the readers field as follows. If readers > 0, there are that many ongoing read-only borrows. If readers $= 0$, there is none, and if readers $= -1$, there is one ongoing read-write borrow. The special value -2 is used to indicate an invalid mpz pointer (such as a pointer to an uninitialized struct). Using the readers field, we can specify two separate borrowing functions. The function get_read_ptr returns a read-only limb pointer, and get_write_ptr returns a pointer with read and write permissions. The writable field has been added to the memory model from Sec. 3.3. It is an immutable boolean field. Once we are done with a pointer obtained using a borrow function, we end the borrow using either the release_reader or the release_writer function. This invalidates the limb pointer and all its aliases. The release functions are erased at extraction (that is, extracted to no-ops). The zones field ensures that the release_* functions cannot be called on the wrong pointers.

Let us now discuss how to update the actual contents of the mpz store. The release_writer function can only be called if the state of the pointer is already consistent with the contents of the store (final precondition). We make it consistent by calling setter functions during the borrow, before calling release_writer. The setter functions are defined in Fig. 4.6.

Let us go over them one by one. The set_size function changes the _mp_size field of an mpz number. It also updates its value in the global store. It is the main way to do so. In order to know the value, it takes a ghost mpn pointer as an extra argument. The zone field is checked to make sure that the pointer is actually the one that was obtained by way of borrowing x. It may seem artificial that the main way to update the value of an mpz pointer in the global store is to update its size. The reason for this design choice is that the size gets updated nearly every time the contents of an mpz pointer are modified. Indeed, as a global invariant, the sizes of mpz pointers are normalized. All the

```
type mpz_memo = abstract {
  mutable abs_value_of : mpz_ptr → int;
  mutable alloc : mpz_ptr → int;
  mutable abs_size : mpz_ptr → int;
  mutable sgn : mpz_ptr → int;
  mutable readers : mpz_ptr → int;
  mutable zones : mpz_ptr → zone;
} invariant { forall p: mpz_ptr.
    0 ≤ abs_size p ≤ alloc p ≤ max_int32
    ∧ (sgn p = 1 ∨ sgn p = -1)
    ∧ 0 ≤ abs_value_of p < power radix (abs_size p)
    ∧ (abs_size p ≥ 1 → power radix (abs_size p - 1) ≤ abs_value_of p) }

function value_of (x:mpz_ptr) (memo: mpz_memo) : int
  = memo.sgn[x] * memo.abs_value_of[x]

val ghost mpz : mpz_memo

val size_of (x: mpz_ptr) : int32                          (* SIZ mpz macro *)
  requires { mpz.readers[x] > -2 }
  ensures { result = mpz.sgn[x] * mpz.abs_size[x] }

let abs_size_of [@extraction:inline] (x: mpz_ptr) : int32 (* ABSIZ mpz macro *)
  requires { mpz.readers[x] > -2 }
  ensures { result = mpz.abs_size[x] }
= abs (size_of x)

val alloc_of (x: mpz_ptr) : int32                         (* ALLOC mpz macro *)
  requires { mpz.readers[x] > -2 }
  ensures { result = mpz.alloc[x] }

val get_read_ptr (x: mpz_ptr) : ptr limb
  requires { mpz.readers[x] ≥ 0 }
  writes   { mpz.readers }
  ensures  { mpz.readers[x] = old mpz.readers[x] + 1 }
  ensures  { forall y. x ≠ y → mpz.readers[y] = old mpz.readers[y] }
  ensures  { value result mpz.abs_size[x] = mpz.abs_value_of[x] }
  ensures  { result.data.length = mpz.alloc[x] }
  ensures  { offset result = 0 ∧ zone result = mpz.zones[x] }
  ensures  { min result = 0 ∧ max result = result.data.length }

val get_write_ptr (x: mpz_ptr) : ptr limb
  requires { mpz.readers[x] = 0 }
  writes   { mpz.readers }
  ensures  { mpz.readers[x] = -1 }
  ensures  { forall y. x ≠ y → mpz.readers[y] = old mpz.readers[y] }
  ensures  { value result mpz.abs_size[x] = mpz.abs_value_of[x] }
  ensures  { result.data.length = mpz.alloc[x] }
  ensures  { offset result = 0 ∧ zone result = mpz.zones[x]
  ensures  { min result = 0 ∧ max result = result.data.length }
  ensures  { writable result }

val release_reader (x: mpz_ptr) (p:ptr limb) : unit
  requires { mpz.zones[x] = zone p ∧ mpz.readers[x] ≥ 1 }
  requires { min p = 0 ∧ max p = p.data.length }
  writes   { mpz.readers, p } (* invalidates p and its aliases *)
  ensures  { mpz.readers[x] = old mpz.readers[x] - 1 }
  ensures  { forall y. y ≠ x → mpz.readers[y] = old mpz.readers[y] }

val release_writer (x: mpz_ptr) (p:ptr limb) : unit
  requires { mpz.zones[x] = zone p ∧ mpz.readers[x] = -1 }
  requires { min p = 0 ∧ max p = p.data.length }
  requires { mpz.abs_value_of[x] = value p mpz.abs_size[x] }
  writes   { mpz.readers, p } (* invalidates p and its aliases *)
  ensures  { mpz.readers[x] = 0 }
  ensures  { forall y. y ≠ x → mpz.readers[y] = old mpz.readers[y] }
```

Figure 4.5: Global store mpz model.

```
predicate mpz_unchanged (x: mpz_ptr) (memo1 memo2: mpz_memo)
  = memo1.readers[x] = memo2.readers[x] ∧
    (memo1.readers[x] > - 2 →
       (memo1.abs_value_of[x] = memo2.abs_value_of[x]
       ∧ memo1.alloc[x] = memo2.alloc[x]
       ∧ memo1.abs_size[x] = memo2.abs_size[x]
       ∧ memo1.sgn[x] = memo2.sgn[x]
       ∧ memo1.zones[x] = memo2.zones[x]))

val set_size (x:mpz_ptr) (sz:int32) (ghost p: ptr limb) : unit
  requires { mpz.zones[x] = zone p }
  requires { mpz.readers[x] = -1 }
  requires { offset p = 0 }
  requires { min p = 0 }
  requires { max p = p.data.length }
  requires { abs sz ≤ p.data.length }
  requires { p.data.length = mpz.alloc[x] }
  requires { sz ≠ 0 → value p (abs sz) ≥ power radix (abs sz - 1) }
  writes  { mpz.sgn, mpz.abs_size, mpz.abs_value_of }
  ensures { forall y. y ≠ x → mpz_unchanged y mpz (old mpz) }
  ensures { mpz.sgn[x] = 1 ↔ 0 ≤ sz }
  ensures { mpz.abs_size[x] = abs sz }
  ensures { mpz.abs_value_of[x] = value p (abs sz) }
  (* ensures size_of x = sz *)

val set_size_0 (x:mpz_ptr) : unit
  requires { -1 ≤ mpz.readers[x] ≤ 0 }
  writes   { mpz.abs_size, mpz.abs_value_of }
  ensures  { forall y. y ≠ x → mpz_unchanged y mpz (old mpz) }
  ensures  { mpz.abs_size[x] = 0 }
  ensures  { mpz.abs_value_of[x] = 0 }

val set_alloc (x:mpz_ptr) (al:int32) : unit
  requires { mpz.abs_size[x] ≤ al }
  requires { -1 ≤ mpz.readers[x] ≤ 0 }
  writes   { mpz.alloc }
  ensures  { forall y. y ≠ x → mpz_unchanged y mpz (old mpz) }
  ensures  { mpz.alloc[x] = al }

val set_ptr (x:mpz_ptr) (p:ptr limb) : unit
  requires { offset p = 0 }
  requires { writable p }
  requires { min p = 0 }
  requires { max p = p.data.length }
  requires { p.data.length = mpz.alloc[x] }
  requires { mpz.readers[x] = 0 ∨ mpz.readers[x] = -1 }
  writes   { mpz.abs_value_of, mpz.zones, mpz.readers }
  ensures  { forall y. y ≠ x → mpz_unchanged y mpz (old mpz) }
  ensures  { mpz.abs_value_of[x] = value p mpz.abs_size[x] }
  ensures  { mpz.readers[x] = -1 }
  ensures  { mpz.zones[x] = zone p }
```

Figure 4.6: Setter functions.

functions from the GMP source code make this assumption. It is also one of the invariants of the global store in our model. Since sizes need to be normalized, in every GMP function that writes into an mpz pointer, a new size is computed (eliminating the leading zero limbs that may have resulted from the computation). In the WhyML code, this translates into the following pattern (where normalize, previously seen in Sec. 4.5.4, is a small helper function that loops through the higher limbs and decreases sz for each leading zero limb).

```
let sz = abs_size_of x in
let p = get_write_ptr x in
... (* computations on p *)
normalize p sz; (* compute normalized size *)
set_size x sz p;
release_writer x p
```

The set_size_0 function is a special case of set_size with relaxed requirements. It exists because some GMP functions set the size of an mpz pointer to 0 without ever looking up its limbs. A typical example is when a subtraction x - x is computed.

The set_alloc function sets the _mp_alloc field of an mpz pointer. It has fewer requirements than set_size, where one could have expected a witness to be required to ensure that the pointer indeed has the reported length. The reason why this is not necessary is that the length is checked again in the preconditions of set_ptr and release_writer.

The set_ptr function is more interesting. It updates the pointer field of an mpz_ptr object. It has a similar specification as release_writer, except that it does not release the borrow: p still has write access on x. It cannot be used if there are read-only borrows of x, but it can be used if there is a read-write one. In that case, the pointer that had previously been borrowed can no longer be released by release_writer, as the zones field has been updated. Typically, it is freed or reallocated.

The main use case for set_ptr is the mpz_realloc function. It takes an mpz_ptr object x and a length sz, and reallocates the data buffer of x so that it has length at least sz (if it was already long enough, it does nothing). The code of a simplified WhyML implementation of mpz_realloc can be found in Fig. 4.7. It has a similar profile as get_write_ptr, in that it returns a pointer that is a read-write borrow of x. The value stored in x is left unchanged. Indeed, when realloc is called to enlarge a pointer, the contents of the original pointer are left unchanged. In the Why3 proof, the last postcondition is proved using the call to the value_sub_frame lemma function at line 26. The lemma essentially states that two arrays that have the same contents have the same value. It can be found in Sec. 4.2.1. It is called on q, which is returned by realloc, and op, which is a *snapshot* of p taken at line 21, before it was passed to realloc.

Creating and clearing mpz pointers

In GMP, the process to obtain a new mpz pointer is as follows. One first declares a mpz_struct, and then calls mpz_init() on its address. The mpz_init function allocates a limb pointer of length 1[4] and initializes the fields of the struct. The mpz_t type is defined as an array of one mpz_struct. This makes the GMP idiom for obtaining a new mpz and initializing it relatively concise.

[4] This was changed in GMP 6.2.0.

```
let wmpz_realloc (x:mpz_ptr) (sz:int32) : ptr limb
  requires { mpz.readers[x] = 0 }
  requires { 1 ≤ mpz.alloc[x] }
  ensures  { forall y. y ≠ x → mpz_unchanged y mpz (old mpz) }
  ensures  { mpz.readers[x] = -1 }
  ensures  { mpz.abs_value_of[x] = value result (mpz.abs_size[x]) }
  ensures  { mpz.zones[x] = zone result }
  ensures  { offset result = 0 }
  ensures  { result.data.length = mpz.alloc[x] }
  ensures  { min result = 0 ∧ max result = result.data.length }
  ensures  { writable result }
  ensures  { mpz.abs_size[x] = old mpz.abs_size[x] }
  ensures  { if sz > old mpz.alloc[x]
             then mpz.alloc[x] = sz
             else mpz.alloc[x] = old mpz.alloc[x]  }
  ensures  { value result (old mpz.abs_size[x]) = old mpz.abs_value_of[x] }
= if sz > alloc_of x
  then begin
    let p = get_write_ptr x in
    assert { forall y. y ≠ x → mpz_unchanged y mpz (old mpz) };
    let ghost op = pure { p } in
    let ghost os = abs_size_of x in
    let q = realloc p sz in
    c_assert (is_not_null q); (* test realloc result and abort if it is NULL *)
    (* q and op have the same contents (postcondition of realloc) *)
    value_sub_frame q.data.elts op.data.elts 0 (int32'int os);
    set_alloc x sz;
    set_ptr x q;
    q
  end
  else get_write_ptr x
```

Figure 4.7: The wmpz_realloc function.

At the end of the program, the `mpz_clear` function is called to avoid memory leaks. It frees the data buffer stored in the `_mp_d` field. The struct itself is allocated on the stack, so it does not need to be explicitly freed. A synopsis is represented below.

```
mpz_t x;
mpz_init(x);
...
mpz_clear(x);
```

The `mpz_init` and `mpz_clear` functions can be modeled without issues. Their specifications are listed below. In order to simplify the specification of `mpz_init`, we exclude the case of `malloc` returning a null pointer. As a result, in the extracted code, the result of `malloc` is checked and the program aborts if a null pointer was returned. In order to account for this, `wmpz_init` is marked `partial` (see Sec. 3.3.5 for details on the `partial` keyword).

```
val partial wmpz_init (p: mpz_ptr) : unit
  requires { mpz.readers[p] = 0 }
  writes  { mpz }
  ensures { forall x. x ≠ p → mpz_unchanged x mpz (old mpz) }
  ensures { mpz.readers[p] = 0 }
  ensures { mpz.abs_value_of[p] = 0 }
  ensures { mpz.abs_size[p] = 0 }
  ensures { mpz.sgn[p] = 1 }
  ensures { mpz.alloc[p] = 1 }
  ensures { mpz.zones[p] ≠ null_zone }

val wmpz_clear (x:mpz_ptr) : unit (* scrambles mpz._[x] *)
  writes { mpz }
  requires { mpz.readers[x] = 0 }
  ensures { forall y. y ≠ x → mpz_unchanged y mpz (old mpz) }
```

One might wonder why the `wmpz_init` function is not simply given a body using a regular let definition. The reason is that the primitives that could be used to implement `wmpz_init` all have preconditions that require their argument to be a valid `mpz` number. Indeed, in our model, the only legal operation that can be performed on an invalid number is to call `wmpz_init` on it. Rather than adding unsafe versions of the primitives and using them to implement `wmpz_init`, it is simpler to simply specify it and replace it by a handwritten C function at extraction.

While the initialization function can be specified in our model, it takes a `mpz_ptr` argument, which is meant to point to an unitialized `mpz_struct`. How do we obtain one in the first place? The model presented above abstracts away the `mpz_struct` type and hides its fields in a global ghost store. However, the extracted code uses the same representation as GMP. In order to obtain new `mpz` objects, the WhyML code needs to contain some instructions that declare the struct and initialize it.

```
type mpz_struct = { ghost addr: mpz_ptr }

val wmpz_tmp_decl [@extraction:c_declaration] () : mpz_struct
  writes  { mpz }
  ensures { old mpz.readers[result.addr] = -2 } (*result is fresh*)
  ensures { forall x. x ≠ result.addr → mpz_unchanged x mpz (old mpz) }
  ensures { mpz.readers[result.addr] = 0 }
  ensures { mpz.alloc[result.addr] = 0 }
  ensures { mpz.abs_size[result.addr] = 0 }
```

```
type mpz_mem = abstract { ptr: mpz_ptr; mutable ok: bool }

val wmpz_struct_to_ptr (x:mpz_struct)
    : (res:mpz_ptr, ghost memo:mpz_mem)
  ensures { res = x.addr }
  ensures { memo.ptr = res }
  ensures { memo.ok }

val ghost wmpz_tmp_clear (x:mpz_ptr) (memo: mpz_mem) : unit
  requires { memo.ok }
  requires { x = memo.ptr }
  requires { -1 ≤ mpz.readers[x] ≤ 0 }
  writes   { mpz, memo }
  ensures  { mpz.readers[x] = -2 }
  ensures  { forall y. y ≠ x → mpz_unchanged y mpz (old mpz) }
```

The `mpz_struct` type is an abstract type that only contains a `mpz_ptr`, which is the address of the struct. The `wmpz_tmp_decl` function creates a new `mpz_struct` object. The corresponding pointer is a valid `mpz_ptr` object with no ongoing borrows. The first two postconditions ensure that the address of the result is distinct from all existing valid `mpz_ptr` objects. They are necessary to specify that the values of existing `mpz` numbers do not change. This function is extracted to a local variable declaration initialized with the struct `{0,0,NULL}`. The `[@extraction:c_declaration]` attribute specifies to the extraction mechanism that this function call is replaced by a local variable definition, so the extraction should check that the new variable does not escape the scope of the function.

The `wmpz_struct_to_ptr` function gets an `mpz_ptr` from the `mpz_struct` it points at. It is extracted to the address operator `&`. It also returns a ghost token of type `mpz_mem`. The purpose of this token is to prevent calling `wmpz_tmp_clear` on `mpz` pointers that were not declared locally. This last function invalidates a local variable of type `mpz_ptr`. It is called at the end of the function in which the latter is declared. Indeed, most `mpz` functions have specifications that state that the `mpz` store is unchanged, except as far as the function arguments are concerned (this is written as an instance of the `mpz_unchanged` predicate). In order to satisfy this, local variables need to be cleaned at the end of the function. As their `readers` field is set to -2, they do not invalidate the `mpz_unchanged` predicate. Cleaning local variables naturally happens in C programs, so there is no need to extract this function and it can be ghost.

4.6.2 A simple function and a GMP bug report

The comparison function is one of the simpler `mpz` functions. It takes two `mpz` numbers `u` and `v`, and returns an integer. The result is 0 if and only if they have the same value. It is positive if `u` is bigger than `v`, and negative otherwise. The algorithm is very simple and the memory model takes up a large chunk of the WhyML implementation. Therefore, a pseudocode would not be meaningful. The WhyML implementation is reproduced in Fig. 4.8. It contains a bug, which is also present in GMP 6.1.2.

Let us first discuss the specification of `wmpz_cmp`. The precondition states that there should be no ongoing read-write borrow of `u` or `v`. It is necessary to perform read-only borrows in the body of the function. This is a restriction

```
let wmpz_cmp (u v:mpz_ptr) : int32
  requires { mpz.readers[u] ≥ 0 ∧ mpz.readers[v] ≥ 0 }
  ensures  { forall w. mpz_unchanged w mpz (old mpz) }
  ensures  { result > 0 → value_of u mpz > value_of v mpz }
  ensures  { result < 0 → value_of u mpz < value_of v mpz }
  ensures  { result = 0 → value_of u mpz = value_of v mpz }
=
  let usize = size_of u in
  let vsize = size_of v in
  let dsize = usize - vsize in
  if dsize ≠ 0 then return dsize;
  let asize = abs usize in
  let up = get_read_ptr u in
  let vp = get_read_ptr v in
  let cmp = wmpn_cmp up vp asize in
  release_reader u up;
  release_reader v vp;
  return (if usize ≥ 0 then cmp else -cmp)
```

Figure 4.8: A buggy comparison function.

compared to GMP. However, the caller could, in principle, release its read-write borrows, then call `wmpn_cmp`, then borrow once again. The first postcondition states that the state of all `mpz` pointers is unchanged at the end of the function. It is necessary to state it because the borrows and releases in the body of the function write into the `mpz` global variable. Therefore, it appears in the `writes` of the function, so we have to state that the original contents are restored. The final three postconditions are the documented postconditions of the algorithm.

We can now discuss the algorithm itself. The core concept is that if the sizes of `u` and `v` are not equal, then the difference of the sizes has the same sign as `u-v`. Indeed, the sizes are signed integers (the sign of the size is the sign of the represented number). Furthermore, all `mpz` numbers are normalized, that is, if `u` has size `usize`, then the number represented by `u` has absolute value greater than or equal to $2^{|\texttt{usize}|-1}$.

Therefore, the algorithm computes the difference `dsize` of the sizes of `u` and `v`. If it is not 0, it returns `dsize`. Otherwise, the numbers have the same size, so the `mpn` function for comparing natural numbers of the same size can be called to compare their absolute values. This result is reversed if both numbers are negative.

However, the algorithm contains a bug. Indeed, the subtraction `usize - vsize` may overflow if `usize` and `vsize` are very large and have opposite sign. In case of overflow, the extracted program invokes undefined behavior. On most machines, the subtraction is computed modulo 2^{32} (or 2^{64}, depending on number sizes) and may return an incorrect result. Note that the bug is unlikely to occur in practice, as the operands need to be several gigabytes long.

I was not able to verify the algorithm presented above, as the subtraction `usize - vsize` has an unprovable precondition stating the absence of overflow. As a result, the verification of WhyMP uncovered the bug. This bug was reported to the GMP developers in February 2020 and has since been fixed[5].

[5] https://gmplib.org/list-archives/gmp-bugs/2020-February/004733.html

4.6.3 Aliasing-related combinatorics

Many mpn functions take parameters that are allowed to be either aliased or separated. WhyML's aliasing restrictions typically force us to define and verify several versions of these functions. Although the tricks from Sec. 3.3.4 collapse them back into a single extracted function, this is still an issue in the WhyML code. This issue is resolved at the mpz layer, where the region-free model allows us to write functions with potentially aliased parameters. However, this makes the body of some mpz functions more convoluted than the original.

The mpz addition functions are a good example of this. Consider the excerpt from the wmpz_add_ui function in Fig. 4.9. It adds an mpz integer u and an unsigned machine integer v, and stores the result in w.

Let us briefly explain the algorithm. There are three main cases, depending on the sign of u. If u is 0, we simply need to set w to v. This is handled in lines 11-20 and poses no particular challenge. Let us now focus only on the case of positive u (the case of negative u is treated in a similar way, swapping additions for subtractions). The first step is to allocate enough space for the new value of w. At most, the sum of u and a limb occupies the size of u plus one limb. At line 25, we call wmpz_realloc, and we have a read-write borrow of w that has sufficient space to store the result.

The actual addition occurs at lines 36-45. There are two cases: either u and w are the same pointer, in which case we call add_1_in_place, or they are not, and we call add_1 (which assumes that the operands are separate in our formalization). This test does not occur in the original C code, which just calls add_1 regardless. In the general case addition (where v is also an mpz integer), the same sort of duplication occurs. A combinatorial explosion occurs, as we have to check equality between u and w as well as between v and w, and there are several cases to handle depending on the (signed) sizes of u and v. Each branch calls a distinct addition or subtraction function, but they are all wrappers around the same core function that accepts aliased pointers, using the trick from Sec. 3.3.4. The wrappers are inlined at extraction. The extracted code ends up needlessly convoluted, with many branches that do very similar things. Some compilers notice this and factor the branches away, but this is not a trivial optimization.

After the carry has been propagated at line 45, wp contains the sum u+v, and it is already the data buffer of w. The only remaining thing to do is some bookkeeping. First, the normalized size of the result is computed. If the addition resulted in a carry, then the result occupies the full abs_usize + 1 limbs. If it did not, then the top limb is empty, and the result occupies abs_usize limbs. It cannot occupy fewer limbs, as u originally occupied abs_usize limbs, and a non-negative integer is added to it. Once the correct size is computed, all that remains to do is storing it in w (line 54) and releasing the borrow.

4.6.4 Aliasing restrictions solved by extra mpz variables

Some mpz functions manipulate mpn pointers directly in ways that cannot be implemented in WhyML due to aliasing restrictions. In these cases, I fell back on the region-free mpz memory model to bypass these restrictions. A good example is the mpz_tdiv_qr function. It is one of several mpz long division functions. It takes four mpz parameters quot, rem, num, and den. As one may

```
let wmpz_add_ui (w u: mpz_ptr) (v: uint64) : unit
  requires { mpz.alloc[w] ≥ 1 ∧ mpz.alloc[u] ≥ 1 }
  requires { mpz.readers[w] = 0 ∧ mpz.readers[u] = 0 }
  requires { mpz.abs_size[u] < max_int32 }
  ensures  { value_of w mpz = old (value_of u mpz + v) }
  ensures  { forall x. x ≠ w → mpz_unchanged x mpz (old mpz) }
  ensures  { mpz.readers[w] = 0 ∧ mpz.readers[u] = 0 }
=
  label Start in
  let usize = size_of u in
  if usize = 0
  then begin
    let wp = get_write_ptr w in
    C.set wp v;
    assert { value wp 1 = v };
    set_size w (if v ≠ 0 then 1 else 0) wp;
    assert { value_of w mpz = v };
    release_writer w wp;
    return;
  end;
  let abs_usize = abs usize in
  assert { 0 < abs_usize ≤ mpz.alloc[u] };
  let uw = mpz_eq u w in
  let ref wsize = abs_usize + 1 in
  let wp = wmpz_realloc w wsize in
  assert { forall x. x ≠ w → mpz_unchanged x mpz (mpz at Start) };
  let ref cy = 0 in
  if usize ≥ 0
  then begin
    begin ensures { value wp abs_usize + power radix abs_usize * cy
                    = old (value_of u mpz) + v }
          ensures { 0 ≤ cy ≤ 1 }
          ensures { uw ∨ mpz.readers[u] = 0 }
          ensures { mpz.readers[w] = -1 }
          ensures { forall x. x ≠ w → mpz_unchanged x mpz (mpz at Start) }
      if uw
      then
        cy ← wmpn_add_1_in_place wp abs_usize v
      else begin
        let up = get_read_ptr u in
        cy ← wmpn_add_1 wp up abs_usize v;
        release_reader u up;
      end
    end;
    C.set_ofs wp abs_usize cy;
    value_tail wp abs_usize;   (* wrapper around value_sub_tail *)
    wsize ← abs_usize + (Limb.to_int32 cy);
    assert { value wp wsize = (value_of u mpz at Start + v) };
    assert { wsize ≠ 0 → value wp wsize ≥ power radix (wsize - 1) };
    end
  else begin
    (* subtraction case, omitted for brevity *)
  end;
  set_size w wsize wp;
  release_writer w wp
```

Figure 4.9: Addition of an mpz integer and a machine integer.

expect, it divides num by den, stores the quotient in quot and the remainder in rem. In the general case, the algorithm simply consists in calling the mpn division and doing the size- and sign-related bookkeeping. The complications arise when some of the parameters are aliased. The specification of the function forbids the case quot == rem. However, num and den are allowed to be aliased with quot or rem. This case is forbidden by the mpn division. The original mpz source code extracts the four mpn pointers of the operands and compares them. If needed, temporary fresh buffers are allocated and passed to the mpn division instead of the original pointer. A relevant excerpt of GMP's source code is reproduced below. Some edits have been made for readability.

```
rp = MPZ_REALLOC (rem, dl);
if (ql <= 0) {
    ...
    return;
  }
qp = MPZ_REALLOC (quot, ql);
np = PTR (num);
dp = PTR (den);
/* Copy denominator to temporary space if it overlaps with the quotient or
     remainder. */
if (dp == rp || dp == qp) {
    mp_ptr tp;
    tp = TMP_ALLOC_LIMBS (dl);
    MPN_COPY (tp, dp, dl);
    dp = tp;
  }
/* Copy numerator to temporary space if it overlaps with the quotient or
     remainder. */
if (np == rp || np == qp) {
    mp_ptr tp;
    tp = TMP_ALLOC_LIMBS (nl);
    MPN_COPY (tp, np, nl);
    np = tp;
  }
mpn_tdiv_qr (qp, rp, 0L, np, nl, dp, dl);
```

We cannot implement this in WhyML as is. First, our C memory model does not have a primitive to test whether two pointers are equal. We could replace the equality tests on mpn pointers by tests on mpz pointers, but it is still not enough. The strong updates dp = tp and np = tp induce reset effects and invalidate the former values of dp and np, preventing us from releasing the corresponding borrows.

The easiest solution would be to move the call to mpn_tdiv_qr inside each if construct, preventing the need for pointer references. The end result would look similar to the addition example from last section, and introduce code and proof duplication.

In order to avoid such code duplication, my WhyML implementation instead creates new mpz variables and performs strong updates on mpz pointers, rather than mpn ones. Since the mpz model has no regions, no reset occurs and the strong updates are no longer a problem. A relevant excerpt of the WhyML implementation can be found in Fig. 4.10. Most of the proof code (assertions, lemmas, and so on) has been omitted for readability.

At lines 16-20, we compute the relevant number sizes. We first eliminate the case where the quotient is 0, in which case calling the mpn division function is not necessary. At line 26, we start preparing for the division proper. We declare

```
let wmpz_tdiv_qr (quot rem num den: mpz_ptr) : unit
  requires { mpz.alloc[num] > 0 ∧ mpz.alloc[den] > 0 }
  requires { mpz.alloc[quot] > 0 ∧ mpz.alloc[rem] > 0 }
  requires { mpz.readers[num] = 0 ∧ mpz.readers[den] = 0
             ∧ mpz.readers[quot] = 0 ∧ mpz.readers[rem] = 0 }
  requires { quot ≠ rem }
  requires { value_of den mpz ≠ 0 }
  requires { mpz.abs_size[num] ≤ max_int32 - 1 }
  ensures  { value_of quot mpz * (old value_of den mpz) + value_of rem mpz
             = old value_of num mpz }
  ensures  { 0 ≤ mpz.abs_value_of rem < old mpz.abs_value_of den }
  ensures  { forall x. x ≠ quot → x ≠ rem → mpz_unchanged x mpz (old mpz) }
  ensures  { mpz.readers[num] = 0 ∧ mpz.readers[den] = 0
             ∧ mpz.readers[quot] = 0 ∧ mpz.readers[rem] = 0 }
=
  let ns = size_of num in
  let ds = size_of den in
  let nl = abs ns in
  let ref dl = abs ds in
  let ref ql = nl - dl + 1 in
  if ql ≤ 0
  then begin
    ... (* quotient is 0, remainder is equal to num *)
    return
  end;
  let ref d' = den in
  let ref n' = num in
  let ghost ref cmemd = any mpz_mem in
  let ghost ref cmemn = any mpz_mem in
  let ghost ref clear_d = false in
  let ghost ref clear_n = false in
  if mpz_eq den rem || mpz_eq den quot
  then begin
    let dp = get_read_ptr den in
    let newd, memd = wmpz_struct_to_ptr (wmpz_tmp_decl ()) in
    d' ← newd;
    cmemd ← memd;
    clear_d ← true;
    let tdp = salloc (UInt32.of_int32 dl) in (* in GMP, either salloc or malloc
      depending on length *)
    set_alloc d' dl;
    wmpn_copyd_sep tdp dp dl;
    set_ptr d' tdp;
    set_size d' ds tdp;
    release_writer d' tdp;
    release_reader den dp;
  end;
  if mpz_eq num rem || mpz_eq num quot
  then begin
    ... (* same treatment for n' as for d' *)
  end;
  assert { d' ≠ quot ∧ d' ≠ rem ∧ n' ≠ quot ∧ n' ≠ rem };
  let qp = wmpz_realloc quot ql in
  let rp = wmpz_realloc rem dl in
  let np = get_read_ptr n' in
  let dp = get_read_ptr d' in
  wmpn_tdiv_qr qp rp 0 np nl dp dl;
  ... (* normalize sizes, handle signs *)
  release_writer rem rp;
  release_writer quot qp;
  release_reader n' np;
  release_reader d' dp;
  ghost (if clear_d then wmpz_tmp_clear d' cmemd);
  ghost (if clear_n then wmpz_tmp_clear n' cmemn);
  return
```

Figure 4.10: Division of relative numbers, in WhyML.

two mutable mpz variables n' and d', initialized at num and den respectively. We also declare four ghost variables at line 28-31. Their semantics are as follows. The denominator that will be passed to the mpn division is the data buffer of d'. If den is aliased to either quot or rem, we cannot have d' == den, so we create a new mpz variable newd and set d' to newd. In this case, the temporary variable newd needs to be cleared at the end of the function, or it would violate the penultimate postcondition. We store the information that newd should be cleared in the clear_d ghost boolean, and the token that allows the user to clear it in cmemd. The variables clear_n and cmemn serve the same role for the numerator.

Note that the operation that clears newd, wmpz_tmp_clear, is a ghost function. Therefore, the flags clear_d and clear_n are themselves allowed to be ghost, and the if statements that clear the temporary variables (lines 62-63) are ghost code as well. Since the mpz_mem tokens are ghost as well, we are allowed to use the any construct at lines 28-29. This constructs tells the proof context that an arbitrary value was assigned. A specification can be added to give extra details about the arbitrary value, although we do not need to do so here. The any construct cannot be extracted. In this case, the use of any is an idiom that allows us to not initialize the ghost references. The arbitrary value is unused anyway. Indeed, the value of cmemd is only used if clear_d is true, in which case cmemd was set to a new value.

The if statement at lines 32-46 sets d' to an mpz pointer that contains the same value as den and is not aliased with quot or rem. If it cannot be den itself, we create newd at line 35. It comes with a write access (readers[newd]= -1). We allocate a temporary pointer tdp on the stack (line 39) and copy den's data to it (line 41). We set tdp as the data pointer of d' at line 42. At line 44, we invalidate tdp and release the borrow on d'. This is allowed because we first used set_ptr. In order to preserve mpz invariants, we also need to set the size and alloc fields of d'. In the end, the if statement is a bit more involved than the original C version. The costly operations (allocating a buffer and copying dp to it) are the same, but our version also allocates a few mpz fields.

The elided if statement at lines 47-50 performs the same treatment for n' as the previous one did for d'. At line 51, we have two pointers d' and n' that are not aliased to quot and rem. Whether they are equal to den and num or not does not need to be known in non-ghost code. At lines 52-55, we get four pointers qp, rp, np, and dp, which are guaranteed to point to separate areas except maybe np and dp (but this is not problematic, as they are read-only). We use realloc rather than get_write_ptr to get qp and rp in order to ensure that they are large enough. At line 56, we can finally call the mpn division. At the end of the function, we release the four data buffers, and invalidate any temporary variables that may exist. At this point, two possibilities exist for the original den pointer. First, it was not aliased with either quot of rem, in which case d' = den, and the test at line 62 is false. Second, it was aliased with one of them, in which case d' is a temporary pointer and den was only ever dereferenced through its alias quot or rem.

4.7 Input/output, string functions

Internally, numbers can be said to be represented in base 2^{64} by GMP. However, GMP needs to interact with client code, which might represent numbers differently. Therefore, it features functions that convert back and forth between GMP's number representation and character strings in which each character encodes a digit in some base.

The work is split between `mpn` and `mpz` functions as follows. The functions `mpn_get_str` and `mpn_set_str` convert back and forth between `mpn` numbers and arrays of unsigned characters that encode numbers in some base b smaller than 256. In C, unsigned characters are integers that range from 0 to 255, so character arrays are well-suited to encoding numbers in base b (we simply use the characters from 0 to $b - 1$).

However, user input typically does not take the form of such an array of unsigned characters. Typically, it is a string such as `"42"` or `"0xdeadbeef"`. In the latter, the string encodes a number in base 16 (as specified by the `"0x"` prefix). The characters are encoded in memory using the ASCII scheme [1], in which the character "4" is represented by the number 52 and "2" by the number 50. The characters represented by the numbers 4 and 2 are unprintable control characters. Clearly, another conversion needs to be performed. It happens in the higher-level `mpz` layer, with which most users interact.

The functions `mpz_get_str` and `mpz_set_str` convert back and forth between human-readable character strings (in bases 2 to 62, where digits above 10 are represented by alphabetic letters) and `mpz` numbers. They perform the conversion between ASCII characters and unsigned characters from 0 to $b - 1$, and call the corresponding `mpn` functions.

The `mpn` functions feature not-quite-trivial arithmetic. The `mpz` functions do not perform many meaningful computations, but they manipulate character strings in ways that pose interesting modeling challenges. For example, the conversion from digit in base b to ASCII character is performed using a hardcoded string literal as a lookup table.

I have verified simplified implementations of all four functions. The original GMP code features divide-and-conquer algorithms and custom optimizations for the common case $b = 10$. Verifying them would have taken too much time. Instead, I verified an implementation that comes from Mini-GMP. Mini-GMP is a single-file, portable, GMP-compatible library that implements a subset of GMP's algorithms. It is "*intended for applications which need arithmetic on numbers larger than a machine word, but which don't need to handle very large numbers very efficiently.*" Its base conversion functions are asymptotically slower, but much simpler. They implement the naive algorithms with an optimization when the base is a power of two.

The section is structured as follows. We introduce some notations in Sec. 4.7.1. We describe the algorithms converting from strings to `mpn` numbers in Sec. 4.7.2, and the ones converting from `mpn` numbers to strings in Sec. 4.7.3. We model the conversion between digits and ASCII characters in Sec. 4.7.4. Finally, we deal with the `mpz` conversion functions in Sec. 4.7.5.

4.7.1 Notations

Before describing the `mpn` base conversion algorithms, let us introduce a few notations. The `mpn` conversion algorithms convert back and forth between `mpn` numbers and big-endian arrays of unsigned characters. The fact that the character arrays are big-endian may seem counter-intuitive, since the `mpn` numbers are represented as little-endian arrays of machine integers. The explanation is that human-readable strings are necessarily big-endian, that is, the first character of a string that represents a number is that number's most significant digit.

Suppose the array s has length n and contains characters that lie between 0 and $b-1$, where $2 \leq b \leq 256$. We represent s as the sequence $s_0 \ldots s_{n-1}$, and we define $\texttt{svalue\_sub}(b, s, i, j) = \sum_{k=i}^{j-1} s_k b^{j-1-k}$ the number encoded by the subarray $s_i \ldots s_{j-1}$ in base b. We sometimes represent it as $\underline{s_i \ldots s_{j-1}}_b$.We also define $\texttt{svalue}(b, s, n) = \texttt{svalue\_sub}(b, s, 0, n) = \sum_{k=0}^{n-1} s_k b^{n-1-k}$ the number encoded by the whole array.

As expected, the `svalue_sub` and `svalue` functions have similar properties to the `value_sub` and `value` functions.

Lemma 9. *Let s a sufficiently long array of characters between* 0 *and* $b-1$*, with* $2 \leq b \leq 256$.

$$\underline{s_0 \ldots s_{n-1}}_b = \underline{s_k \ldots s_{n-1}}_b + b^{n-k} \underline{s_0 \ldots s_{k-1}}_b \qquad [\texttt{svalue\_sub\_concat}]$$

$$\underline{s_0 \ldots s_{n-1}}_b = \underline{s_1 \ldots s_{n-1}}_b + b^{n-1} s_0 \qquad [\texttt{svalue\_sub\_tail}]$$

$$\underline{s_0 \ldots s_n}_b = s_n + b \cdot \underline{s_0 \ldots s_{n-1}}_b \qquad [\texttt{svalue\_sub\_head}]$$

4.7.2 From base b to base β

Let us now verify Mini-GMP's `mpn_set_str` function. It reads from an array of characters in some base b and converts the result to an `mpn` number. The algorithm is split into two main cases: a general case `set_str_other`, and a special case `set_str_bits`. The latter is an optimization for the case where `b` is a power of two.

Special case: $b = 2^i$

Let us first go over `set_str_bits` (Alg. 25). It takes four parameters: an array s of characters, its length n, a small integer i, and a buffer r. It converts $\underline{s_0 \ldots s_{n-1}}_{2^i}$ to base β and stores it in r. Our WhyML implementation takes an extra ghost parameter z, which is a lower bound on the required length of the buffer r. This way, specifications such as "there is sufficient space in r to store the result" can be expressed concisely. For readability, the pseudocode is specialized for the case $\beta = 2^{64}$.

The algorithm is straightforward. Indeed, 2^i and β are both powers of two. If r was a contiguous array of bits, all we would have to do would be to concatenate the characters from s (and reverse their order, to account for endianness). Instead, r is split into 64-bit limbs, and the algorithm needs to account for the case of a character being split between two limbs.

The algorithm is a simple loop that reads characters from s one by one. The loop invariants are as follows.

Algorithm 25 Converting from base 2^i to base $\beta = 2^{64}$.

```
Require: $0 < n \leq 2^{32} - 1$
Require: $\texttt{valid}(r, z) \wedge 0 < z$
Require: $\texttt{valid}(s, n)$
Require: $1 \leq i \leq 8$
Require: $2^{i \cdot n} \leq \beta^z$                                ▷ There is enough space in $r$.
Require: $\forall k.\ 0 \leq k < n \implies 0 \leq s_k < 2^i$                  ▷ $s$ is in base $2^i$.
Ensure: $0 \leq \texttt{result} \leq z$
Ensure: $\texttt{svalue}(2^i, s, n) = \texttt{value}(r, \texttt{result})$
Ensure: $\texttt{result} > 0 \implies r[\texttt{result} - 1] > 0$          ▷ The result is normalized.
function SET_STR_BITS($r$, ghost $z, s, n, i$)
    $l \leftarrow 0$
    $h \leftarrow 0$        ▷ Currently occupied bits in $r[l-1]$. 0 means no space left.
    $j \leftarrow n$
    while $j > 0$ do
        $j \leftarrow j - 1$
        if $h = 0$ then                          ▷ No space left in $r[l-1]$, go to $r[l]$.
            $r[l] \leftarrow s_j$
            $l \leftarrow l + 1$
            $h \leftarrow i$
        else                                         ▷ Add as much of $s_j$ as fits.
            $r[l-1] \leftarrow r[l-1] + (s_j \ll h)$
            $h \leftarrow h + i$
            if $h \geq 64$ then
                $h \leftarrow h - 64$
                if $h > 0$ then                            ▷ It did not fit entirely.
                    $r[l] \leftarrow s_j \gg (i - h)$          ▷ Add the rest to the next limb.
                    $l \leftarrow l + 1$
    NORMALIZE($r, l$)
    return $l$
```

1) $0 \leq h < 64$

2) $0 \leq j < n$

3) $\texttt{value}(r, l) = \underline{s_j \dots s_{n-1}}_{2^i}$

4) $h > 0 \implies r[l-1] < 2^h$

5) $0 \leq l \leq z$

6) $j > 0 \implies i \cdot (n - j) = \begin{cases} 64l & \text{if } h = 0 \\ 64(l-1) + h & \text{if } h > 0 \end{cases}$

The first two are self-explanatory and the third states that the base conversion is correct. The fourth states how the variable h tracks how much space is taken in the current limb. The last two are needed to prove that we are not writing outside the bounds of r. In the last invariant, the left-hand side of the equality is the number of bits already consumed from s, and the right-hand side is the number of bits currently written in r.

The initialization is trivial to prove for all six invariants. Let us show that they are maintained through a loop iteration. Instantiating `svalue_sub_tail` yields that we need to increase $\texttt{value}(r, l)$ by $2^{i \cdot (n-j)} s_j$ to validate invariant **3)**.

There are three main cases. Either $h = 0$, $0 < h \leq 64 - i$, or $h > 64 - i$. In the first case, $r[l-1]$ is full, so we write in $r[l]$ and increment l. This increases $\texttt{value}(r, l)$ by $\beta^l s_j$ (for the old value of l), which equals $2^{i \cdot (n-j)} s_j$ as per the last invariant.

The second case is similar. At line 12, we add $2^h s_j$ to $r[l-1]$. There is no carry, because $r[l-1]$ was less than 2^h, and $2^h s_j \leq 2^h(2^i - 1) \leq \beta - 2^h$. Furthermore, the total quantity added to r is $\beta^{l-1} 2^h s_j = 2^{i \cdot (n-j)} s_j$ as per the last invariant. The variable h is incremented by i modulo 64 and the end result fails the condition at line 16.

In the third case, we have $h + i > 64$ and there is not enough room for s_j in $r[l-1]$. We split s_j into two halves s_l and s_h, with $s_j = s_l + 2^{64-h} s_h$ and $s_l < 2^{64-h}$. At line 12, we add $2^h s_l$ to $r[l-1]$. Indeed, we compute $2^h s_j = 2^h s_l + \beta s_h = 2^h s_l \mod \beta$. The addition does not overflow for the same reason it did not in the previous case. At line 17, we write s_h into $r[l]$. The total quantity added to r is $\beta^{l-1} 2^h s_l + \beta^l s_h = \beta^{l-1} 2^h (s_l + 2^{64-h} s_h) = 2^{i \cdot (n-j)} s_j$.

This concludes the proof that invariant **3)** is maintained. The only other non-trivial invariant is **5)**, that is, $l \leq z$. The reason it holds is the following. Without loss of generality, consider an iteration of the loop where l increases. The preconditions gives us $i \cdot n \leq 64z$. Since $j \geq 1$, per the last invariant, if $h = 0$ we have $64l \leq 64z - i$ so $l < z$. If $h > 0$, then similarly, $64(l-1) + h \leq 64z - i$, so $64l \leq 64z - (h - (64 - i))$. Since l increases in this loop iteration, we have $h > 64 - i$, so $l < z$. Since l increases by at most 1, we still have $l \leq z$ at the end of the iteration.

After the loops, invariants imply $\texttt{svalue}(2^i, s, n) = \texttt{value}(r, l)$. However, l may not satisfy the third postcondition (for example, if the first few characters of s were zeroes). We call the `normalize` function, which iterates through the high limbs of l and decreases l for each leading zero limb. This leaves $\texttt{value}(r, l)$ unchanged and validates the last postcondition.

General case

Let us now go over the general case algorithm. The naive algorithm for converting a number s from base b to base β is as follows. We iterate over the digits of s starting from the most significant one. For each digit w, we multiply our currently accumulated result by b and add w. This algorithm is correct, but requires a long multiplication per digit in s. Mini-GMP's algorithm is a slightly optimized version of the naive algorithm. In order to reduce the number of long multiplications, the digits are handled by batches of e, where b^e fits into a limb. For each batch of e digits, we apply the naive algorithm to convert them to a limb, and then we multiply the accumulated result by b^e and add that limb. Let us formalize this algorithm. The pseudocode can be found in Alg. 26.

Algorithm 26 Conversion from base b to base $\beta = 2^{64}$: general case.

```
Require: $0 \le n < 2^{32} \wedge \texttt{valid}(s, n)$
Require: $0 < z \wedge \texttt{valid}(r, z)$
Require: $2 \le b \le 256$
Require: $b^n \le \beta^z$                                  ▷ There is enough space in $r$.
Require: $\forall k.\ 0 \le k < n \implies 0 \le s_k < b$           ▷ $s$ is in base $b$.
Require: $B = b^e$
Require: $B < \beta \le B \cdot b$          ▷ $B$ is the largest power of $b$ that fits in a limb.
Ensure: $\texttt{value}(r, \texttt{result}) = \texttt{svalue}(b, s, n)$
Ensure: $1 \le \texttt{result} \le z$
Ensure: $\texttt{svalue}(b, s, n) > 0 \implies r[\texttt{result} - 1] > 0$
Ensure: $\texttt{svalue}(b, s, n) = 0 \implies \texttt{result} = 1$
function SET_STR_OTHER($r$, ghost $z, s, n, b, B, e$)
    $k \leftarrow 1 + (n - 1 \mod e)$
    $w \leftarrow s_0$
    $j \leftarrow 1$
    while $k > 1$ do
        $k \leftarrow k - 1$
        $w \leftarrow w \cdot b + s_j$
        $j \leftarrow j + 1$
    $r[0] \leftarrow w$
    $l \leftarrow 1$
    while $j < n$ do
        $w \leftarrow s_j$
        $j \leftarrow j + 1$
        for $k = 1$ to $e - 1$ do
            $w \leftarrow w \cdot b + s_j$
            $j \leftarrow j + 1$
        $c \leftarrow$ MUL_1($r, r, l, B$)
        $c' \leftarrow$ ADD_1($r, r, l, w$)
        $c \leftarrow c + c'$
        if $c > 0$ then                                           ▷ $l < z$.
            $r[l] \leftarrow c$
            $l \leftarrow l + 1$
    return $l$
```

The function takes the same parameters as the previous one, plus two additional ones. In addition to the base b, it also takes an exponent e, and a precomputed $B = b^e$. B is the largest power of b that fits in a limb.

The algorithm splits s in batches of e characters. However, e may not divide n. The first loop (lines 5-9) handles the odd batch of fewer than e characters. Its main invariant is that w stores the information from the first j characters: $w = \texttt{svalue}(b, s, j)$. The number of iterations is chosen such that after the loop, e divides $n - j$.

The main loop handles the rest of the characters from s. Each iteration handles a batch of e characters. The loop invariants are as follow:

1) $0 \leq j \leq n$

2) $0 \leq l \leq z$

3) $\texttt{svalue}(b, s, j) > 0 \implies r[l-1] > 0$

4) $\texttt{svalue}(b, s, j) = 0 \implies l = 1$

5) $\texttt{value}(r, l) = \texttt{svalue}(b, s, j)$

6) $n - j \equiv 0 \mod e$

The first four invariants are clearly initialized. The first loop sets up the last two. Let us justify that they are maintained through a loop iteration. The `for` loop at lines 14-16 computes $\texttt{svalue\_sub}(b, s, j, j + e)$ into w (the loop invariant is $w = \texttt{svalue\_sub}(b, s, j, j + k)$). The lemma `svalue_sub_concat` gives $\texttt{svalue}(b, s, j + e) = w + B \cdot \texttt{svalue}(b, s, j)$. This quantity is computed at lines 17-18. More precisely, we have $\texttt{value}(r, l) + \beta^l(c + c') = \texttt{svalue}(b, s, j)$, with j having been increased by e. At lines 19-22, if there is a carry, we write it into $r[l]$ and increment l. This validates invariant **5)**. Invariant **6)** is also valid, since j was increased by e. The remaining non-trivial invariant is the second, that is, $l \leq z$. Without loss of generality, we assume $\texttt{svalue}(b, s, j) > 0$. Since $j \leq n$, $\texttt{svalue}(b, s, j) < b^j \leq \beta^z$. Since $r[l-1] > 0$, we have $\texttt{value}(r, l) \geq \beta^{l-1}$. Therefore, $l - 1 < z$.

4.7.3 From base β to base b

The conversion from `mpn` numbers to arrays of characters in base b is also split into two algorithms. Let us first handle the case where b is a power of two.

Special case $b = 2^i$

When b is a power of two, say 2^i, the algorithm is simple. Each batch of i bits in the input corresponds to a character in base 2^i in the output. The function `get_str_bits` (Alg. 27) takes an `mpn` number u, its length n, and an exponent i, converts $\texttt{value}(u, n)$ to base 2^i and writes the result in the array s. Much like in the previous case, we add a ghost parameter z that is a lower bound on the length of s, which makes the specification more concise. Note that z is in bits this time.

The algorithm first computes the length of the input number in base 2^i. There are c leading zeros in $u[n-1]$, and $u[n-1] > 0$ per the preconditions.

Algorithm 27 Converting from base $\beta = 2^{64}$ to base 2^i.

```
Require: $1 \le n \land \texttt{valid}(u, n)$
Require: $1 \le i \le 8$
Require: $0 \le z \land \texttt{valid}(s, \lceil \frac{z}{i} \rceil)$
Require: $\texttt{value}(u, n) < 2^z$                    ▷ There is enough space in $s$.
Require: $u[n-1] > 0$
Require: $64n + 7 < 2^{32}$
Ensure: $\forall k.\ 0 \le k < \texttt{result} \implies 0 \le s_k < 2^i$
Ensure: $\texttt{svalue}(2^i, s, \texttt{result}) = \texttt{value}(u, n)$
Ensure: $s_0 > 0$
function GET_STR_BITS($s$, ghost $z, u, n, i$)
    $c \leftarrow$ COUNT_LEADING_ZEROS($u[n-1]$)
    $e \leftarrow 64n - c + (i - 1)$
    $l \leftarrow e/i$                                 ▷ Length of the result in base $2^i$.
    $b \leftarrow 1 \ll i$
    $j \leftarrow l$                                   ▷ Current position in $s$.
    $k \leftarrow 0$                                   ▷ Current position in $u$.
    $h \leftarrow 0$                   ▷ Number of bits consumed in current limb.
    while $j > 0$ do
        $j \leftarrow j - 1$
        $g \leftarrow u[k] \gg h$
        $h \leftarrow h + i$
        if $h \ge 64$ then      ▷ We consumed the full limb, go to the next one.
            $k \leftarrow k + 1$
            if $k < n$ then
                $h \leftarrow h - 64$
                $g \leftarrow g + (u[k] \ll (i - h))$      ▷ Make sure $g$ has at least $i$ bits.
        $s_j \leftarrow (g \mod b)$
    return $l$
```

Therefore, we have $2^{64n-c-1} \leq \texttt{value}(u, n) < 2^{64n-c}$, that is, u is $64n - c$ bits long. We compute $l = \lceil \frac{64n-c}{i} \rceil$, which is the length of that number in base 2^i.

The main loop reads i bits from u at each iteration, converts them to a character in base 2^i, and writes it into s. Since the final length of s is already known, it can start with the least significant bits. The loop invariants are as follows. They use the function `bitvalue` from Sec. 4.3.3 to talk about values of u at the bit granularity. Much like in the `set_str_bits` function, h tracks how many bits were already consumed for the current limb.

1) $0 \leq k \leq n$

2) $0 \leq j \leq l$

3) $j > 0 \implies i(l - j) = 64k + h$

4) $0 \leq h < 64 \vee k = n$

5) $\forall x.\ j \leq x < l \implies 0 \leq s_x < 2^i$

6) $\texttt{svalue\_sub}(2^i, s, j, l) = \begin{cases} \texttt{bitvalue}(u, 64k + h) & \text{if } j > 0 \\ \texttt{value}(u, n) & \text{if } j = 0 \end{cases}$

The initialization of the invariants is trivial. Let us justify that they are maintained through a loop iteration. At line 11, g contains the top $64 - h$ bits of u, that is, $\texttt{value}(u, k+1) = \texttt{bitvalue}(u, 64k + h) + 2^{64k+h}g$. We first notice that if $k = n$, $j = 0$. Indeed, at the beginning of the loop iteration where k is incremented to n, we have $i(l - j) = 64(n - 1) + h$ and $h + i \geq 64$, so $64n - i \leq i(l - j)$. Since $il \leq e < 64n + i$, we have $ij < 2i$, so $j = 1$ at the start of the loop, and it is decreased to 0 at line 10. This validates the first invariant. The remaining non-trivial ones are the last two. There are two main cases.

The easy case is $h + i \leq 64$, that is, there are at least i bits in g. In this case, the test at line 13 fails. We split g into $g' + 2^i g''$, with $0 \leq g' < 2^i$. We have $\texttt{bitvalue}(u, 64k{+}h{+}i) = \texttt{bitvalue}(u, 64k{+}h){+}2^{64k+h}g'$. We write g' into s_{j-1} at line 18, and we have $\texttt{svalue\_sub}(2^i, s, j - 1, l) = \texttt{svalue\_sub}(2^i, s, j, l) + 2^{i(l-j)}g'$. Since $i(l - j) = 64k + h$ per the invariant hypotheses, this validates the last invariant if $j > 0$. If $j = 0$, then we have read the last non-zero bits of u, that is, $64k{+}h{+}i \geq 64n{-}c$. Therefore, $\texttt{value}(u, n) = \texttt{bitvalue}(u, 64k{+}h{+}i)$ and the last invariant is also validated. Invariant **5)** is also valid, since g' is between 0 and 2^i.

In the second case, $h + i > 64$. At line 11, g only has $64 - h$ bits. If $k = n$, then j is decreased to 0 at line 10 as explained earlier, and $\texttt{value}(u, n) = \texttt{svalue\_sub}(2^i, s, 1, l) + 2^{i(l-j)}g$. At line 18, since $g < 2^i$, we write g into s_0 and the invariants are valid. If $k < n$, we get the remaining bits from $u[k+1]$. More precisely, at line 11, we have $\texttt{value}(u, k+1) = \texttt{bitvalue}(u, 64k + h) + 2^{64k+h}g$. Let us split $u[k + 1]$ into $g' + 2^{h+i-64}g''$, with $0 \leq g' < 2^{h+i-64}$. We have $\texttt{bitvalue}(u, 64k + h + i) = \texttt{value}(u, k + 1) + \beta^{k+1}g' = \texttt{bitvalue}(u, 64k + h) + 2^{64k+h}(g + 2^{64-h}g')$. Since $2^{64k+h} = 2^{i(l-j)}$, we need to compute $g + 2^{64-h}g'$ and write it in $s[j - 1]$. At line 16, the new value of $i - h$ is $i - (h_0 + i - 64) = 64 - h_0$, where h_0 is the old value of h. The addition at line 17 computes $g + 2^{64-h_0}g' + 2^i g''$ modulo β. There is no carry because $g < 2^{64-h_0}$ (in fact, this addition is implemented as a bitwise *or* in the original code). At line 18, the reduction modulo b leaves $g+2^{64-h_0}g'$ to be written in s_{j-1}, which validates the last two invariants.

Let us now assume that the two invariants are valid. The only thing left to check is that $s_0 > 0$, and this is a consequence of the fact that we computed l precisely. Indeed, $\texttt{value}(u, n) \geq 2^{64n-c-1} \geq 2^{i(l-1)}$. Moreover, $\texttt{svalue}(2^i, s, l) = 2^{i(l-1)}s_0 + \texttt{svalue\_sub}(2^i, s, 1, l)$ and $\texttt{svalue\_sub}(2^i, s, 1, l) < 2^{i(l-1)}$, so we must have $s_0 > 0$.

General case for one limb

The general case conversion from base β to base b uses an auxiliary function that converts one limb to base b. The algorithm is the naive one: the input is repeatedly divided by b, and each remainder is a digit in base b. The only issue is that the least significant digits are computed first, and it is not trivial to compute the size of the output in advance. Since the output should be big-endian, where to write the least significant digits? The solution adopted by Mini-GMP is to produce a little-endian array first, and then reverse it.

We define yet another value function to represent little-endian arrays. If s is an array $s_0 \dots s_{n-1}$ of characters whose values are between 0 and $b-1$, we define $\texttt{svalue\_le}(b, s, n) = \sum_{i=0}^{n-1} s_i b^i$. It is essentially a generalized version of the `value` function where the base is specified, rather than fixed to β. The same lemmas apply, such as $\texttt{svalue\_le}(b, s, k+l) = \texttt{svalue\_le}(b, s, k) + b^k\texttt{svalue\_le}(b, s+k, l)$ (`svalue_le_sub_concat`).

The function `limb_get_str` (Alg. 28) converts a limb to a little-endian array in base b using an optimized version of the naive algorithm. The gist of it is as follows. We want to divide the limb by b repeatedly. However, GMP's division algorithm normalizes the divisor so that it is greater than $\beta/2$, and then computes its pseudo-inverse. Rather than letting it perform the same normalization and pseudo-inverse computation many times, the caller normalizes b in advance and supplies a pseudo-inverse. As a result, the function has seven parameters: the input limb w, the output array s and its (ghost) length z, the normalized base d, its pseudoinverse v, and a shift h such that $d = 2^h b$, as well as b itself, which is not used by the code but is passed as a ghost parameter to simplify the specification.
Let us call w_0 the initial value of w. The algorithm is a simple loop. Its invariants are the following.

1) $0 \leq w < \beta$

2) $0 \leq i \leq z$

3) $w > 0 \implies i < z$

4) $\forall k.\ 0 \leq k < i \implies 0 \leq s_k < b$

5) $w = 0 \implies (i = 0 \vee s_{i-1} > 0)$

6) $w_0 = \texttt{svalue\_le}(b, s, i) + b^i w$

Once again, the initialization is trivial. The invariants also clearly imply the postconditions. Let us prove that the invariants are maintained through a loop iteration. At lines $4-5$, we compute l and m such that $l + \beta m = 2^h w$. At

Algorithm 28 Converting a limb to base b.

```
Require: $2 \le b \le 256$
Require: $d \ge \beta/2$
Require: $d = 2^h b \wedge 0 \le h \le 63$
Require: $v = \textsc{reciprocal}(d)$
Require: $0 < z \wedge \texttt{valid}(s, z)$
Require: $w < b^z$                                  ▷ There is enough room in $s$.
Ensure: $\texttt{svalue\_le}(b, s, \texttt{result}) = \texttt{old}\ w$
Ensure: $0 \le \texttt{result} \le z$
Ensure: $\forall k.\ 0 \le k < \texttt{result} \implies 0 \le s_k < b$
Ensure: $\texttt{result} > 0 \implies s_{\texttt{result}-1} > 0$
 1: function LIMB_GET_STR($s, \texttt{ghost}\ z, w, d, v, h, \texttt{ghost}\ b$)
 2:     $i \leftarrow 0$
 3:     while $w > 0$ do
 4:         $m \leftarrow w \gg (64 - h)$
 5:         $l \leftarrow w \ll h$                           ▷ $l + \beta m = 2^h w$.
 6:         $q, r \leftarrow \textsc{div2by1\_inv}(m, l, d, v)$    ▷ $qd + r = 2^h w$.
 7:         $s_i \leftarrow r \gg h$
 8:         $w \leftarrow q$
 9:         $i \leftarrow i + 1$
10:     return $i$
```

line 6, we divide $l + \beta m$ by d (using the precomputed pseudoinverse v). We obtain q and r such that $qd + r = 2^h w$, with $0 \le r < d$. Moreover, $d = 2^h b$. Therefore, we have $2^h w = 2^h bq + r$. Therefore, 2^h divides r, so the logical right shift at line 7 does not overflow. Let us define r' such that $r = 2^h r'$. We have $w = bq + r'$. After writing r' in s_i, we have $w_0 = \texttt{svalue\_le}(b, s, i) + b^i(bq + r') = \texttt{svalue\_le}(b, s, i + 1) + b^{i+1}q$, which validates the last invariant. Let us check the other five. The first two are trivial. The third is a consequence of the fact that at the end of the iteration, $b^{i+1}q \le w_0 < b^z$, so if $q > 0$, then $i + 1 < z$. The fourth invariant is valid because $r < 2^h b$, so $r' < b$. The fifth is valid because at the beginning of the loop iteration, $w > 0$, so if $q = 0$, then $r = 2^h w \ge 2^h$, so $r' \ge 1$.

General case

We are now ready to review the general case algorithm (Alg. 29). The main idea is still the naive one, that is, repeatedly dividing the input by b. However, in order to reduce the number of long divisions, the algorithm is instead repeatedly dividing by some large power of b that fits into a limb. Each chunk is then handled by the `limb_get_str` function, which only performs short divisions.

The function takes seven parameters: the output array s and its ghost size, the base b, its largest power B that fits in a limb, the corresponding exponent e, the input u, and its size n. Let us call $U = \texttt{value}(u, n)$ the value of the input. The function first precomputes the necessary information to later call `limb_get_str`: the normalized base d, its pseudoinverse v, and the shift h such that $d = 2^h b$. Then, if there is more than one limb in the input, it is repeatedly divided by B until only one limb remains. The invariants of the main loop are as follows. The ghost variable i tracks the number of loop iterations, which

Algorithm 29 Conversion from base β to base b.

```
Require: $1 \le n \wedge \texttt{valid}(u, n)$
Require: $u[n-1] > 0$
Require: $0 < z \wedge \texttt{valid}(s, z)$
Require: $\texttt{value}(u, n) < b^{z-1}$        ▷ There is enough space for $u$, plus one extra
    character.
Require: $B = b^e$
Require: $B < \beta \le B \cdot b$        ▷ $B$ is the largest power of $b$ that fits in a limb.
Ensure: $0 \le \texttt{result} < z$        ▷ The conversion leaves one free cell at the end.
Ensure: $\texttt{svalue}(b, s, \texttt{result}) = \textbf{old } \texttt{value}(u, n)$
Ensure: $\forall k.\ 0 \le k < \texttt{result} \implies 0 \le s_k < b$        ▷ The result is in base $b$.
Ensure: $s_0 > 0$
function GET_STR_OTHER($s$, ghost $z, b, B, e, u, n$)
    $h \leftarrow$ COUNT_LEADING_ZEROS($b$)
    $d \leftarrow b \ll h$        ▷ Normalized base for division.
    $v \leftarrow$ INVERT_LIMB($d$)        ▷ Precomputed pseudoinverse.
    $l \leftarrow 0$
    $k \leftarrow n$
    if $k > 1$ then
        $t \leftarrow$ ALLOC($n$)
        ghost $i \leftarrow 0$
        while $k > 1$ do
            $w \leftarrow$ DIVREM_1($t, u, k, B$)
            COPYI($u, t, k$)
            $k \leftarrow k - ($if $u[k-1] = 0$ then $1$ else $0)$
            $o \leftarrow$ LIMB_GET_STR($s + l, e, w, d, v, h, b$)
            $l \leftarrow l + o$
            while $o < e$ do        ▷ If $w$ was small, pad with zeroes.
                $s_l \leftarrow 0$
                $l \leftarrow l + 1$
                $o \leftarrow o + 1$
            $i \leftarrow i + 1$
    $o \leftarrow$ LIMB_GET_STR($s + l, z - 1 - l, u[0], d, v, h, b$)  ▷ Read the last limb.
    $l \leftarrow l + o$        ▷ $U = \texttt{svalue\_le}(b, s, l)$.
    $j \leftarrow 0$
    while $2j + 1 < l$ do        ▷ Reverse $s$.
        $x \leftarrow s_j$
        $s_j \leftarrow s_{l-j-1}$
        $s_{l-j-1} \leftarrow x$
        $j \leftarrow j + 1$
    return $l$
```

makes the invariants a bit more concise.

1) $1 \leq k \leq n$

2) $n - k \leq i$

3) $l = ie$

4) $\forall j.\ 0 \leq j < l \implies 0 \leq s_j < b$

5) $0 \leq l < z$

6) $u[k-1] > 0$

7) $U = \texttt{svalue\_le}(b, s, l) + b^l \cdot \texttt{value}(u, k)$

Let us justify that they are maintained. At lines $11 - 12$, u is divided by B. The remainder is stored in w and the quotient is copied back in u. As a result, we have $U = \texttt{svalue\_le}(b, s, l) + b^l w + b^{l+e}\texttt{value}(u, k)$. This is left unchanged by line 13, which normalizes u by decreasing k if needed. We cannot have both $u[k-1] = 0$ and $u[k-2] = 0$ before normalization. Indeed, $\texttt{value}(u, k)$ was greater than β^{k-1} (by invariant hypothesis **6)**), and it was divided by $B < \beta$, so it is still greater than β^{k-2}. Therefore, u is indeed normalized after line 13, which validates invariant **6)**. At line 14, `limb_get_str` is called. Per its postcondition and the concatenation lemma on `svalue_le`, we have $U = \texttt{svalue\_le}(b, s, l + o) + b^{l+e}\texttt{value}(u, k)$. If $o < e$, the inner loop pads s with zeroes until e characters have been written. The size l is increased while $\texttt{svalue\_le}(b, s, l)$ remains unchanged. The main invariant of the inner loop is $U = \texttt{svalue\_le}(b, s, l) + b^{l-o+e}\texttt{value}(u, k)$. After the inner loop, the last invariant is maintained. This justifies the last two invariants. The first four are clearly valid by construction. Let us now justify the fifth, that is, $l < z$. By precondition, $U < b^{z-1}$, and the last loop invariant yields $b^l \cdot \texttt{value}(u, k) \leq U$. Since $\texttt{value}(u, k) > 0$ (invariant **6)**), we have $l < z - 1$.

After the loop, we have $k = 1$, and we have $U = \texttt{svalue\_le}(b, s, l) + b^l u[0]$ per the last invariant. The function `limb_get_str` is called one last time. (The reason it could not be done inside the loop is that the result may be greater than e.) We have $b^l u[0] \leq U < b^{z-1}$, so $u[0] < b^{z-1-l}$, which validates the relevant precondition of `limb_get_str`. After the call, the concatenation lemma on `svalue_le` yields $U = \texttt{svalue\_le}(b, s, l)$. Furthermore, the postcondition yields $o \leq z - 1 - l$, so after line 22, we have $l < z$, which validates the first postcondition. We also have $s_{l-1} > 0$. All that remains to do is reverse the array s, which is done at lines $24 - 27$.

Let us call s' the state of the array s after the loop. For all $0 \leq i < l$, we have $s'_i = s_{l-1-i}$. If l is even, each cell of the array is swapped with the mirrored cell exactly once. If l is odd, the middle cell is never swapped. More precisely, the main invariant is as follows:

$$\forall i.\ 0 \leq i < l, s'_i = \begin{cases} s_i & \text{if } j \leq i < l - j \\ s_{l-1-i} & \text{if } 0 < j \text{ or } l - j \leq i. \end{cases}$$

After s is reversed into s', we have $s'_0 > 0$ and all the postconditions are valid, as $\texttt{svalue}(b, s', l) = \texttt{svalue\_le}(b, s, l) = U$.

4.7.4 ASCII conversions

The mpn algorithms from the previous section convert back and forth between mpn numbers (in base β) and arrays of digits in base b, with $2 \le b \le 256$. However, they are not in a printable format. The mpz conversion algorithms convert back and forth between ASCII characters and the digits they encode. The code of the mpz conversion functions is not particularly interesting from a verification standpoint, but their specifications are.

The mpz conversion functions support bases ranging from 2 to 62. For bases between 2 and 35, digits greater than 9 are encoded by the lowercase letters a-z. For bases greater than 36, digits from 10 to 35 are encoded by the uppercase letters A-Z, and digits from 36 to 61 are encoded by the lowercase letters a-z. Negative bases from -36 to -2 are also supported. They are similar to their positive counterparts, but encode digits from 10 to 35 as uppercase letters. Therefore, one can get the conversion functions to output a hexadecimal number in lowercase, such as "0xdeadbeef" by setting the base to 16. Setting the base to -16 would output "0xDEADBEEF", in uppercase.

Let us first briefly present a Why3 formalization of the conversion between digit and ASCII character. We define the ASCII codes of the alphanumeric characters using string constants, as follows.

```
constant digitstring : string = "0123456789"
constant lowstring : string = "abcdefghijklmnopqrstuvwxyz"
constant upstring : string = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

axiom numcodes:
      forall i. 0 ≤ i < 10 →
             code (get digitstring i) = code (get "0" 0) + i

axiom lowcodes:
      forall i. 0 ≤ i < 26 →
             code (get lowstring i) = code (get "a" 0) + i

axiom upcodes:
      forall i. 0 ≤ i < 26 →
             code (get upstring i) = code (get "A" 0) + i

axiom code_0: code (get "0" 0) = 48
axiom code_a: code (get "a" 0) = 97
axiom code_A: code (get "A" 0) = 65
axiom code_minus: code (get "-" 0) = 45
```

Using these values, we define logical functions that serve as a specification of the conversion between an unsigned integer d (with the type unsigned char) and an ASCII character c. The formalization appears in Fig. 4.11. These functions cannot be used directly in WhyML programs, but they appear in specifications. As they are logical functions, they need to be total. However, we are not interested in specifying the conversions for non-alphanumeric characters. The text_to_num returns the bogus value -1 when applied to an invalid character. Similarly, we are not interested in specifying what num_to_text does to numbers that are not between 0 and 61. Therefore, we define a bogus value dummy_char as some character whose ASCII code is -1. (The chr function is defined as the reciprocal of the code function. It is total, but its specification only covers the case where its argument is between 0 and 255.)

As a sanity check, we verify a lemma that states that text_to_num is re-

```
constant dummy_char = chr (-1)

constant numlowstring : string = "0123456789abcdefghijklmnopqrstuvwxyz"

function num_to_lowercase_text (d:uchar) : char
= if 0 ≤ d < 36
  then get numlowstring d
  else dummy_char

constant numupstring : string = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"

function num_to_uppercase_text (d:uchar) : char
= if 0 ≤ d < 36
  then get numupstring d
  else dummy_char

constant numuplowstring : string
  = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

function num_to_bothcase_text (d:uchar): char
= if 0 ≤ d < 62
  then get numuplowstring d
  else dummy_char

function num_to_text (base:int) (d:uchar) : char
= if 0 ≤ base ≤ 36
  then num_to_lowercase_text d
  else if 36 < base ≤ 62
  then num_to_bothcase_text d
  else if -36 ≤ base
  then num_to_uppercase_text d
  else dummy_char

function text_to_num_onecase (c:char) : int
= if (get "0" 0) ≤ c ≤ (get "9" 0)
  then c - (get "0" 0)
  else if (get "a" 0) ≤ c ≤ (get "z" 0)
  then c - (get "a" 0) + 10
  else if (get "A" 0) ≤ c ≤ (get "Z" 0)
  then c - (get "A" 0) + 10
  else -1

function text_to_num_bothcase (c:char) : int
= if (get "0" 0) ≤ c ≤ (get "9" 0)
  then c - (get "0" 0)
  else if (get "a" 0) ≤ c ≤ (get "z" 0)
  then c - (get "a" 0) + 36
  else if (get "A" 0) ≤ c ≤ (get "Z" 0)
  then c - (get "A" 0) + 10
  else -1

function text_to_num (base:int) (c:char) : int
= if - 36 ≤ base ≤ 36
  then text_to_num_onecase c
  else text_to_num_bothcase c
```

Figure 4.11: Converting back and forth between ASCII characters and digits.

ciprocal to num_to_text. The proof involves a number of intermediate lemmas and is about 100 lines long.

```
let lemma tnt (base:int) (d:uchar)
  requires { -36 ≤ base ≤ 62 }
  requires { 0 ≤ d < abs base }
  ensures  { text_to_num base (num_to_text base d) = d }
= ...
```

Using the conversion functions, we can define the value of a given array of characters in some base b as a mathematical integer. In addition to the alphanumeric characters that encode the digits in base b, we also recognize minus signs at the beginning of the array. The body of the abs_value_sub_text function calls strlen, when one may have expected an extra length argument. Indeed, C strings are unlike mpn numbers in that they are typically not accompanied by an integer that specifies their length. Instead, they are terminated by the null character, encoded by the number 0. Their length is the position of the first null character. The function strlen from the <string.h> header is used to compute the length. We declare a function strlen and give it a formal specification. At extraction, it can be replaced by its C counterpart.

```
let rec ghost function abs_value_sub_text (b:int) (s:map int char) (n m: int)
    : int
  variant { m - n }
= if n < m
  then text_to_num b s[m-1] + b * abs_value_sub_text b s n (m-1)
  else 0

function abs_value_text (b:int) (s:map int char) (ofs:int) : int
  = abs_value_sub_text b s ofs (ofs + strlen s ofs)

function value_text (b:int) (s:map int char) (ofs:int) : int
  = if Char.(=) s[ofs] minus_char
    then - abs_value_text b s (ofs + 1)
    else abs_value_text b s ofs

function strlen (s:map int char) (ofs:int) : int

axiom strlen_def:
  forall s ofs i. 0 ≤ i
  → (forall j. 0 ≤ j < i → code s[ofs + j] ≠ 0)
  → code s[ofs + i] = 0
  → strlen s ofs = i

axiom strlen_invalid:
  forall s ofs. (forall i. 0 ≤ i → code s[ofs + i] ≠ 0)
  → strlen s ofs < 0

predicate valid_string (p: ptr char)
  = strlen p.data.elts (offset p) ≥ 0
    ∧ valid p (1 + strlen p.data.elts (offset p))

val strlen (p: ptr char) : uint32
  requires { valid_string p }
  ensures  { result = strlen p.data.elts (offset p) }
```

It may seem strange that there are two separate strlen declarations. The reason is that the behavior of strlen is specified by the strlen_def and strlen_invalid axioms, and only logical functions can be referred to in the logic. Therefore, we need to first declare a logical function strlen, specify its behavior using the axioms, and finally declare a program function (also called `strlen`) and state that it behaves like the logical function.

4.7.5 Base conversions in `mpz`

We have now laid out the tools needed to verify the `mpz` conversion functions. Rather than a full algorithm, we simply show the specifications and the sections of code that perform the conversion between ASCII character and digit.

From string to `mpz`

Let us start with the `mpz_set_str` function (Fig. 4.12). It takes a character string and a base (between 2 and 62) as inputs, and converts the string into an `mpz` number if it is well-formed, in which case it returns 0. If the string is ill-formed, the function returns −1. Note that our WhyMP formalization of this function is not quite feature-complete. The mini-GMP version accepts 0 as a base, in which case it tries to infer the base by checking the beginning of string for a prefix such as `"0x"` or `"0b"`. It also accepts and ignores whitespace in the string. Our version does not accept any whitespace, and requires the base to be explicitly specified.

The well-formedness of a string is defined using the `string_in_base` predicate below. A string is well-formed if it contains only characters that are valid digits in its base (with the exceptions of the null terminator and a possible minus sign) and has at least one digit.

```
predicate text_in_base (b:int) (t: map int char) (n m:int)
  = forall i. n ≤ i < m → 0 ≤ text_to_num b t[i] < b

predicate string_in_base (b:int) (s:map int char) (ofs: int)
  = (text_in_base b s ofs (ofs + strlen s ofs) ∧ strlen s ofs > 0)
    ∨ (s[ofs] = minus_char
       ∧ text_in_base b s (ofs + 1) (ofs + strlen s ofs)
       ∧ strlen s ofs > 1)
```

The function computes the length of the input string, allocates an array of the same length, and converts the string to digits in base b. It then calls the relevant `wmpn_set_str` function to convert them to base β. The snippet that converts a character to a digit has the same structure as the logical function `text_to_num`, which was designed for this exact purpose. Therefore, the assertion `digit = text_to_num base c` is easy to prove.

From `mpz` to string

The `mpz_get_str` function (Fig. 4.13) converts an `mpz` number to a character string. Bases from 2 to 62 are supported, as well as bases from −36 to −2 (which are used to output uppercase letters). If the base is between −1 and 1, the function still accepts it but treats it as 10. The logical function `effective` is used to specify this behavior. In our version of the algorithm, the output buffer needs to be already allocated and have enough space to store the output. In order to specify the length needed for the output buffer, we use an extra ghost parameter `sz`. This is a missing feature compared to the mini-GMP version, which allocates a sufficiently large buffer for the user if they pass a null pointer.

```
let wmpz_set_str (r: mpz_ptr) (sp: ptr char) (base: int32) : int32
  requires { valid_string sp }
  requires { (strlen sp.data.elts (offset sp)) * 8 + 63 ≤ max_int32 }
  (* avoids an overflow when computing the size of the result *)
  requires { mpz.readers[r] = 0 }
  requires { mpz.alloc[r] ≥ 1 }
  requires { 2 ≤ base ≤ 62 }
  ensures  { forall x. x ≠ r → mpz_unchanged x mpz (old mpz) }
  ensures  { mpz.readers[r] = 0 }
  ensures  { result = 0 →
             value_of r mpz = value_text base sp.data.elts (offset sp) }
  ensures  { -1 ≤ result ≤ 0 }
  ensures  { result = 0 ↔ string_in_base base sp.data.elts (offset sp) }
=
  ...
  let dp : ptr uchar = salloc (strlen sp) in
  let sign = if C.get sp = minus_char then 1 else 0 in
  let ref spi = C.incr sp sign in
  let ref digit : uchar = 0 in
  let ref dn = 0 in
  while (C.get spi ≠ zero_char) do
    ...
    invariant { abs_value_sub_text base (pelts sp)
                                   (offset sp + sign) (offset spi)
                = svalue_sub base (pelts dp) 0 dn }
    ...
    if 36 < base
    then begin
      if code zero_num ≤ code c && code c ≤ code nine_num
      then digit ← UChar.of_int32 (code c - code zero_num)
      else if code small_a ≤ code c && code c ≤ code small_z
      then digit ← UChar.of_int32 (code c - code small_a + 36)
      else if code big_a ≤ code c && code c ≤ code big_z
      then digit ← UChar.of_int32 (code c - code big_a + 10)
      else begin
        (* invalid string *)
        digit ← UChar.of_int32 base
      end
    end
    else begin
      ...
    end;
    assert { 0 ≤ digit < base → digit = text_to_num base c }
    if digit ≥ UChar.of_int32 base
    then begin
      (* invalid string, abort *)
      set_size_0 r;
      return -1
    end;
  ...
  done;
  ...
  (* call wmpn_set_str *)
  return 0;
```

Figure 4.12: Conversion from string to mpz number.

```
function effective (b:int) : int = if (abs b < 2) then 10 else abs b

let wmpz_get_str (sp: ptr char) (ghost sz:int32)
                 (base:int32) (u: mpz_ptr) : ptr char
  requires { valid sp sz }
  requires { writable sp }
  requires { 2 ≤ sz }
  (* need two extra spaces to store the minus sign and the null terminator *)
  requires { mpz.abs_value_of[u] < power (effective base) (sz-2) }
  requires { mpz.readers[u] = 0 }
  requires { 64 * mpz.abs_size[u] + 7 ≤ max_int32 }
  (* precondition of get_str_bits *)
  requires { -36 ≤ base ≤ 62 }
  ensures { result = sp }
  ensures { valid_string sp }
  ensures { string_in_base (effective base) sp.data.elts (offset sp) }
  ensures { forall x. mpz_unchanged x mpz (old mpz) }
  ensures { value_text (effective base) sp.data.elts (offset sp)
            = value_of u mpz }
=
  let digits =
    if base > 36
    then "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
    else if base ≥ 0
    then "0123456789abcdefghijklmnopqrstuvwxyz"
    else "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ" in
  ...
  let ref i : int32 = 0 in
  if size_of u < 0
  then begin
    C.set_ofs sp 0 minus_char;
    i ← i+1
  end;
  (* call to wmpn_get_str_bits or wmpn_get_str_other,
     digits written in sp *)
  for j = i to sn - 1 do
    ...
    let cj = UChar.of_char (C.get_ofs sp j) in
    let dc = String.get digits (UChar.to_int32 cj) in
    assert { cj = text_to_num base dc };
    C.set_ots sp j dc;
    ...
  done;
  ...
  C.set_ofs sp sn zero_char; (* null terminator *)
  return sp;
```

Figure 4.13: Conversion from mpz number to string.

The function calls the relevant `mpn_get_str` function and writes the result in `sp`. It then converts the characters in `sp`, and adds a null terminator. The conversion is performed using the alphabetical string `digits` as a lookup table. This is exactly how the logical function `num_to_text` was defined, so the assertion that the conversion was correct is easily proved.

4.8 Comparing WhyMP and GMP

WhyMP is designed as a verified clone of the GMP library. It implements a subset of GMP's interface, and it is intended to serve as a drop-in replacement for GMP. Nonetheless, it is not a perfect clone of GMP. Many GMP functions are simply not implemented yet. Furthermore, some code changes had to be made for the sake of the verification process, although the intent was for WhyMP to be almost as efficient as GMP. This section presents the main differences between the libraries (Sec. 4.8.1), as well as some performance benchmarks (Sec. 4.8.2).

4.8.1 Compatibility, code changes

WhyMP is largely compatible with GMP. Both libraries represent the numbers in the same way. WhyMP's generated C functions have the same signature as their GMP counterparts. Using the appropriate headers, one can either use WhyMP as a replacement for GMP, or mix and match the libraries. However, there are two potential sources of incompatibility. First, WhyMP currently lacks some genericity. It represents limbs as 64-bit numbers, so it should not be interfaced with a 32-bit GMP. However, the `mpz` layer hides this detail of number representation from the user, so WhyMP can still be used as a replacement for a 32-bit GMP as long as the user code does not mix both libraries and interacts only with `mpz` numbers. Second, GMP supports custom memory handlers passed by the user, while WhyMP performs its allocations using `malloc`. Therefore, in order to interface both libraries, one needs to use GMP's default memory handler as well.

Aliasing

Due to WhyML's aliasing restrictions, some WhyMP functions in the `mpn` layer have restricted specifications compared to their GMP counterparts. For example, GMP's division allows the numerator to be identical to the remainder, while WhyMP's does not. In principle, this restriction can be overcome using the methods from Sec. 3.3.4. I have done so for WhyMP's long addition and subtraction. Due to time and effort constraints, this was not done yet for the following functions: addition and subtraction of a limb to an `mpn` number, multiplication of a limb by an `mpn` number, addition or subtraction of the product of a limb and an `mpn` number to another `mpn` number (`addmul_1` and `submul_1`), and all variants of division. This does not make these functions incorrect, they are simply unverified in the aliased case.

The main reason why I have not yet taken the time to implement this is that these constraints are hidden by the `mpz` wrappers, which do have the exact same signature and specification as their GMP counterparts. Therefore, this issue does not affect the users who only interact with WhyMP or GMP through the `mpz` layer, which is a very common case.

However, in order to account for the calling restrictions of the mpn layer, the code of many mpz functions had to be modified. Examples can be found in Sec. 4.6.3, 4.6.4, and 4.7. These changes make the code less idiomatic, but should not have an overly large impact on performance. Indeed, they mostly add operations whose costs do not scale with the length of the numbers involved, and few costly operations such as copies of mpn numbers.

Missing algorithms

Some differences between GMP and WhyMP do have a meaningful impact on performance. First and foremost, WhyMP implements fewer algorithms than GMP. For example, GMP features more than ten different multiplication algorithms, while WhyMP only has Toom-2, Toom-2.5, and the base case algorithm. As a result, for very large inputs, GMP uses algorithms that are much more asymptotically efficient. Similarly, GMP features a subquadratic divide-and-conquer division algorithm in addition to the optimized schoolbook algorithm that WhyMP implements.

When reimplementing GMP's algorithms in WhyMP, I strove to preserve all the optimizations present in GMP's implementation. However, I gave up on a small number of optimizations that required a very large amount of effort and did not seem critical in terms of performance. For example, GMP features a relatively complex dedicated squaring algorithm. It uses it instead of the regular multiplication algorithm when the operands are known to be equal. It also sometimes checks operands for pointer equality, and calls the squaring function if they are equal. The latter pattern would be hard to implement in the mpn layer WhyMP due to the aliasing restrictions of our memory model. As a result, I gave up on verifying the dedicated squaring algorithm, assuming that there were not that many contexts where operands are known to be equal. In retrospect, this was a mistake. The square function is explicitly used by GMP's divide-and-conquer square root, as well as the critical loop of the modular exponentiation algorithm. Their performances are worse in WhyMP as a result. I also underestimated the performance difference between the dedicated squaring algorithm and the generic multiplication.

Primitives

A sizable difference between the internals of GMP and WhyMP lies in the underlying arithmetic primitives. Both WhyMP and GMP rely on the availability of a multiplication of one limb by one limb returning two limbs, as well as a division of two limbs by one limb. For most architectures, with default compilation options, GMP implements these primitives, as well as many basic functions, with native assembly routines. These assembly routines are not terribly complicated, but not trivial either. In order to minimize the trusted computing base, WhyMP instead relies on the 128-bit support from C compilers. For example, here is the primitive that WhyMP uses for division.

```
uint64_t div64_2by1 (uint64_t ul, uint64_t uh, uint64_t d)
  { return (((uint128_t)uh << 64) | ul) / d; }
```

The code uses the type uint128_t, which is not a standard C99 type. However, it is arguably simple enough to trust it.

4.8.2 Benchmarking

The following benchmarks show how GMP and WhyMP compare on three benchmarks: multiplication, square root, and a primality test. More precisely, three variants of GMP and three variants of WhyMP are tested. The difference between these variants is the implementation of the underlying primitives. Indeed, there is a large performance gap between the performance of native assembly routines and the performance of routines written in C, even with the 128-bit extension. Therefore, the direct comparison between WhyMP and default GMP is not that meaningful, so we measure more timings to give a better view of the performances.

First, GMP is also compiled without support for assembly, which means that only the generic C code is compiled. GMP without assembly and WhyMP are not exactly in the same ballpark though, since they do not use the same primitive operations for doing a $64 \times 64 \to 128$ multiplication and a 128-by-64 division. Indeed, in assembly-free GMP, these are implemented in C using only 64-bit operations, which are much less efficient than the 128-bit ones that WhyMP gets from the compiler.

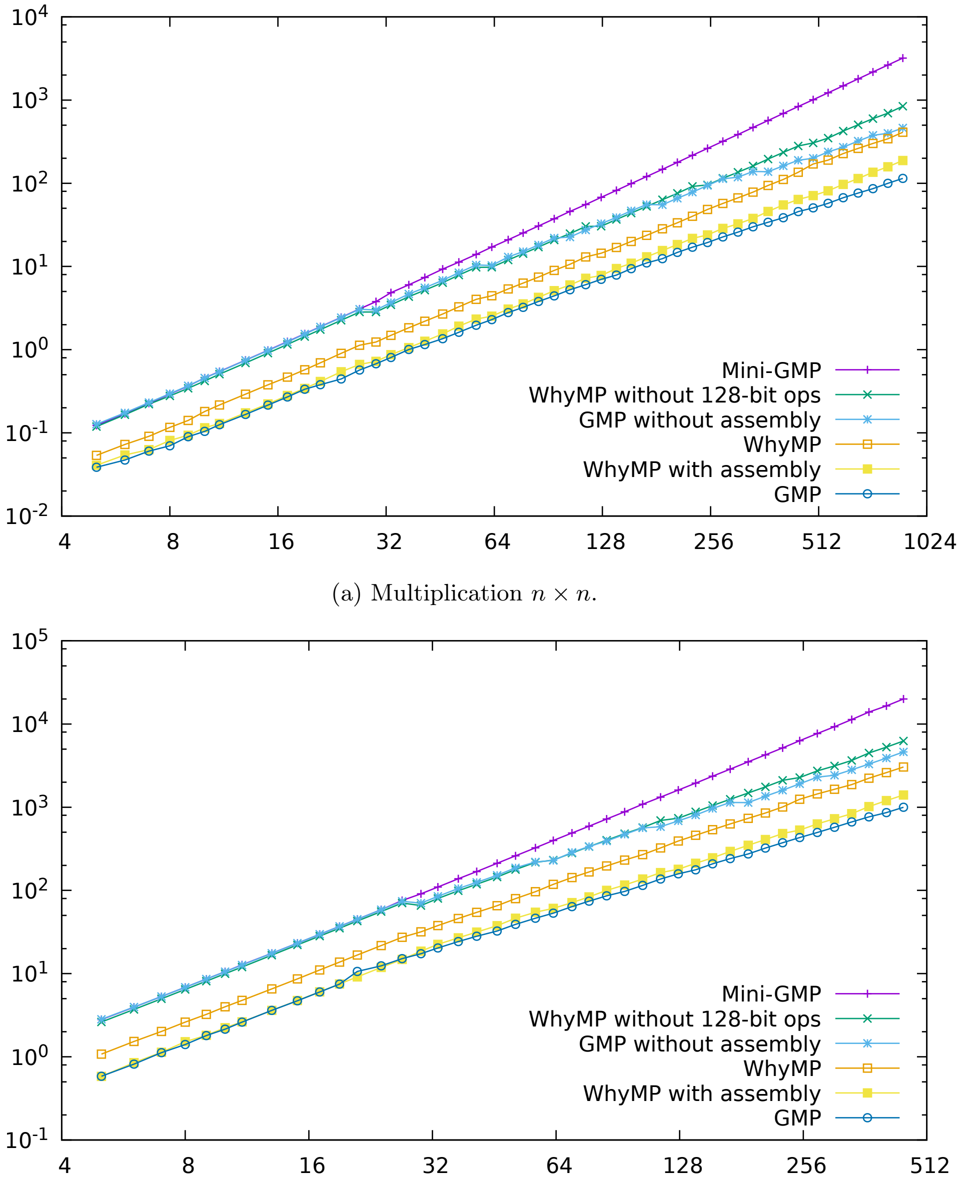
Second, to measure the impact of these two primitives, WhyMP is also compiled in a way such that their 128-bit implementation is replaced by the 64-bit one from GMP without assembly.

Third, the timings of Mini-GMP are measured. It uses the same kind of implementation as GMP without assembly for the two primitives above, that is, it uses only 64-bit operations.

Finally, in an attempt to measure the performance of higher-level algorithms in isolation, some low-level `mpn` functions of WhyMP are replaced by their respective GMP counterparts, as these functions are typically written in assembly. Those functions are `add_n` (resp. `sub_n`), which computes the sum (resp. difference) of equally-sized `mpn` numbers; `add` and `sub`, for `mpn` numbers with different sizes; `mul_1`, which multiplies an `mpn` number by a single limb; `addmul_1` (resp. `addmul_2`), which multiplies an `mpn` number by a single limb (resp. a two-limb number), and then accumulates the product into the destination; and `submul_1`, which accumulates the opposite of the product. Note that we could have replaced a lot more functions of WhyMP by their assembly counterparts from GMP, including rather complicated ones, such as division by two-limb numbers. Instead, we chose to focus on a few simple functions, so as to not blow the trusted code base out of proportions, which would defeat the point of formally verifying an arithmetic library.

The version of GMP is 6.1.2, which was the most recent release when this work started. The benchmarks are executed on an Intel Xeon E5-2450 at 2.50 GHz. All the libraries are compiled using GCC 8.3.0 using the options selected by GMP, that is, "`-O2 -march=sandybridge -mtune=sandybridge -fomit-frame-pointer`".

Figures 4.14 to 4.16 show the timings obtained on the various benchmarks. On every figure, abscissas are the number of 64-bit limbs, while ordinates are the time in microseconds. All the figures are in log-log scale, so that the asymptotic complexity is apparent. Performance-wise, the general ordering of the plots is the same on every figure: GMP is the fastest, then comes WhyMP with GMP's assembly primitives, then WhyMP, then GMP without assembly support, then WhyMP without 128-bit support, and finally Mini-GMP is the slowest.



(a) Multiplication $n \times n$.

(b) Multiplication $n \times 24n$.

Figure 4.14: Timings for multiplication.

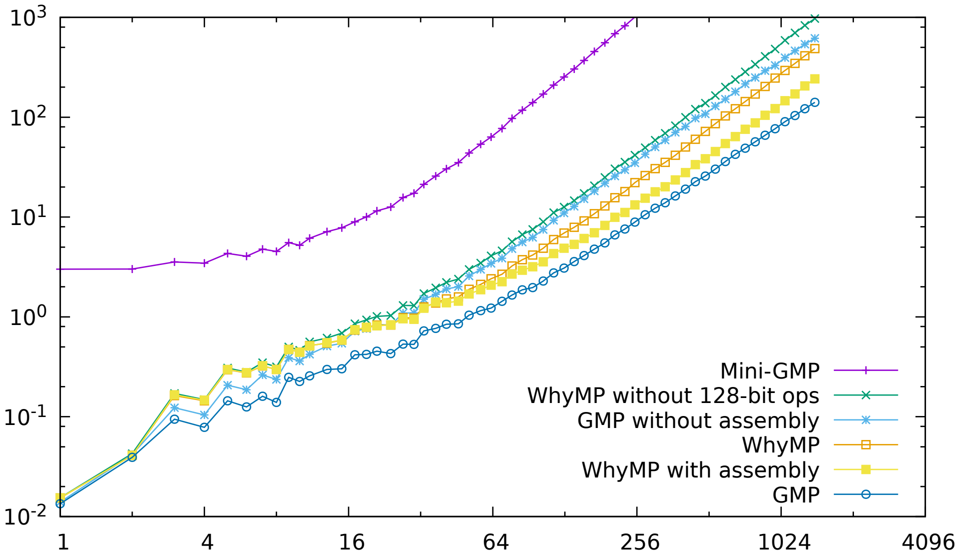


Figure 4.15: Timings for square root.

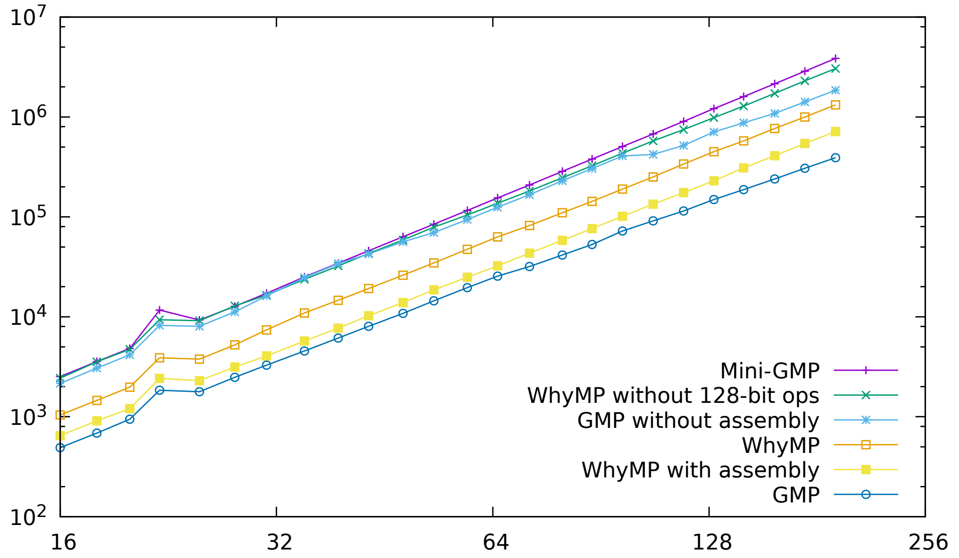


Figure 4.16: Timing for Miller-Rabin.

Multiplication (Fig. 4.14)

The first benchmark simply tests multiplication for various sizes of `mpn` numbers, so as to exercise both the base-case multiplication as well as Toom-Cook algorithms. Two cases are tested: equal-sized inputs, and $n \times 24n$ unbalanced inputs.

The unbalanced case tests the algorithmic differences between WhyMP and GMP. Indeed, WhyMP performs 16 calls to `toom_32`, which results in 64 $\frac{n}{2} \times \frac{n}{2}$ multiplications, while GMP performs 12 calls to `toom_42`, which results in 60 $\frac{n}{2} \times \frac{n}{2}$ multiplications. Due to the extra cost of interpolation for `toom_42`, WhyMP hardly suffers from not having `toom_42` at this level of unbalance.

Comparing the plots of Mini-GMP, WhyMP without 128-bit support, and GMP without assembly, makes it apparent when the libraries switch to different algorithms. Mini-GMP sticks with the quadratic schoolbook algorithm, while WhyMP and GMP switch to `toom_22` around $n = 30$, and then GMP switches to `toom_33` around $n = 60$. Starting around $n = 170$ (`toom_44` for GMP), the lack of higher variants of Toom-Cook in WhyMP becomes noticeable, as the library becomes progressively slower with respect to GMP. For $n \leq 170$, WhyMP is at most twice as slow as GMP, and when replacing the primitive operations with the assembly ones from GMP, the slowdown does not exceed 20%. The smaller n is, the smaller the slowdown, down to about 5% for $n \leq 20$.

Square root (Fig. 4.15)

The second benchmark tests the square root for various sizes of `mpn` numbers. GMP's algorithm performs a long division, so WhyMP greatly suffers from featuring only the schoolbook division, despite using the same divide-and-conquer square-root algorithm as GMP. This makes WhyMP with assembly about 50% slower than GMP for $n \leq 600$. Without assembly, WhyMP is twice as slow for $n \leq 90$, and thrice as slow for $n \leq 600$. As for Mini-GMP, its poor performance (up to ×150 times slower for $n \leq 600$) can be explained by the fact that its multiplication algorithm is quadratic, as well as the use of a converging sequence $y_{n+1} = (x/y_n + y_n)/2$, rather than a dedicated square root algorithm.

Miller-Rabin's primality test (Fig. 4.16)

The third benchmark implements Miller-Rabin's primality test for number sizes commonly encountered in cryptography applications. This is a simple implementation inspired from GMP's one. It exercises the `mpz` layer as well as the modular exponentiation. Note that the modular exponentiation used in WhyMP is just a wrapper over `mpn_powm`, so it supports neither even moduli nor negative exponents, contrarily to `mpz_powm`. WhyMP is 110% slower than GMP for $n \leq 28$, and 140% slower for $n \leq 60$. With assembly primitives, the slowdown is less than 30% for $n \leq 60$.

Evaluation

Overall, two factors have a large impact on performance: the complexity of the algorithms, and the quality of the underlying arithmetic primitives. On large numbers, WhyMP's multiplication and division fall behind even that of the assembly-free version of GMP when the latter switches to asymptotically better algorithms. In all other cases, the algorithms are similar enough that the primitives seem to be the deciding factor. What we conclude from this is that WhyMP's algorithms are close enough to the original that most of the performance difference comes from the primitives written in handwritten assembly, at least for smaller inputs.

4.9 Evaluation, perspectives

WhyMP was developed concurrently with my Why3 model of the C language, as well as the extraction mechanism from WhyML to C. It served both as a proof of concept of the verification approach they enable and as a way to evaluate these tools. Let us go over some lessons learned from the development of WhyMP, related work, and possible lines of future work.

4.9.1 Proof effort and lessons learned

As far as I am aware, WhyMP is the largest existing Why3 development, as well as one of the first to span more than a few thousand lines of code. The WhyMP sources currently total about 22 000 lines of WhyML code. A more detailed breakdown can be found in Fig. 4.17. The automated solvers that I used are Alt-Ergo (2.0.0, 2.2.0 and 2.3.0), CVC3 (2.4.1), CVC4 (1.5, 1.6 and 1.7), Z3 (4.5.0 and 4.6.0), the E prover (1.9.1-001 and 2.0), and the Gappa tool (version 1.3.5). Replaying the proofs takes about an hour.

Each of the provers is necessary, in the sense that there are goals that only this prover can check. It would be plausible to use only the most recent version of each prover (the old versions are mostly used for historical reasons), but even this would take a non-trivial amount of work that I was unable to do due to time constraints. This naturally raises the question of the soundness of the solvers, especially in the suspicious cases where only one prover manages to check a goal. All these provers are indeed in the trusted code base. This issue is somewhat mitigated by Why3's `bisect` feature. After one solver manages to prove a goal, `bisect` can be used to reduce the logical context to the smallest one that still leaves the goal provable, using the succesful solver as an oracle to shave off useless premises. In most cases, several other provers also manage to prove the goal using the reduced context.

Among the 22 000 lines of code in the WhyMP sources, only about 8 000 of them are program code. The other 14 000 are made up of specifications and (mostly) assertions. It is worth mentioning that this ratio is not particularly efficient as Why3 proofs go. As a point of comparison, the Why3 sources contain a repository of about 200 example proofs. It contains about 400 WhyML files. Discounting WhyMP, it totals about 18 000 lines of program code and 27 000 lines of proof code.

This relative inefficiency is not particularly surprising. Indeed, most of the proofs of complex arithmetic facts in WhyMP were performed using many explicit proof hints in the form of very long assertions, sometimes over a hundred lines long. This also explains how time-consuming the verification process was. The next chapter focuses on explaining why the automated provers were not well suited to automatically proving these facts, and describing the efforts that

comparison	100
addition	1000
subtraction	1000
mul (naïve)	700
mul (Toom)	2400
division	4500
helper lemmas	300
reflection	1700
shifts	1000
square root	1600
exponentiation	1700
base conversions	2000
`mpz` layer	3600
utilities	200

Figure 4.17: Proof effort in lines of code per WhyMP function.

were made to automate some of these proofs by implementing and verifying a dedicated decision procedure in WhyML.

Some proofs went much better than expected. An example of this is the proof of the fixed-point square root algorithm. Although the algorithm is best understood as an instance of Newton iteration, the concept did not need to appear in the WhyML development. Furthermore, the Gappa tool was extremely well-suited to proving the required inequalities. Most of them were automatically discharged while I did not understand the full details of why they were true and would not have been able to write a pen-and-paper proof for them.

The proof served as a meaningful test of my region-based C memory model. I initially expected the GMP functions to not be overly complex in terms of aliasing, making such a region-based model a natural choice. This turned out to not be entirely accurate. Throughout the verification process, I ended up needing to add more and more ways for the memory model to circumvent the aliasing constraints of WhyML. I still do believe that the choice of a region-based memory model was correct. Indeed, I was very well served by the automated alias tracking in all the situations where there were no complex aliases in the `mpn` layer. By contrast, the formalization of the `mpz` layer used a region-free memory model, where the aliasing needed to be handled directly in the logic. This was a forced choice, as the functions of the `mpz` layer are very permissive in terms of aliasing of the parameters. However, the lack of a region-based model was harshly felt. A significant part of the proof effort was spent solving the frame problem manually, by instantiating lemmas that stated that the values of most `mpz` numbers had not changed between two program points. In the end, I still believe that a region-based memory model was the right tool to verify a subset of GMP in Why3, by making the best use of the latter's type system. However, it turned out that GMP was not as convenient a case study as I initially believed.

4.9.2 Related work

We have used Why3 to formally verify GMP's integer arithmetic layer. We obtain a verified and efficient C library. Previous work generally did not deal with a large number of highly optimized algorithms. As far as we know, this work is the first formal verification of a comprehensive and efficient arbitrary-precision integer library. Let us now list some examples of existing verifications of arithmetic functions or libraries. Many of these examples were already listed in the introduction, but we can now discuss them in more detail and compare them to this work.

Myreen and Curello produced a verified arbitrary-precision integer arithmetic library using the HOL4 theorem prover and separation logic [77]. Their library covers the four basic arithmetic operations, but not the square root. They do not attempt to produce highly efficient code. As a result, the algorithms they proved are simpler and less efficient than the optimized ones that we proved. For example, their multiplication algorithm is the schoolbook one. However, their verification goes all the way down to x86 machine code, using formally verified proof-generating compilers and decompilers to do part of the proof on a higher-level implementation. Using these tools, they also managed to avoid most proofs involving pointer reasoning. The total proof effort is about 6 000 lines. Their proof effort per algorithm is roughly similar to ours despite them using an interactive tool.

Affeldt used Coq to verify a binary extended GCD algorithm implemented in a variant of MIPS assembly [4], as well as the functions it depends on, such as addition, subtraction, and halving. The work encompasses both signed and unsigned integer arithmetic. It uses GMP's number representation and a memory model based on separation logic. The author verifies the algorithm in a pseudo-code language and proves a forward simulation relation between the pseudo-code and the MIPS assembly code to prove the latter's correctness. It makes some simplifying assumptions, such as requiring that the operands share the same length. The GCD algorithm that is verified is not trivial, but it is much less involved and efficient than the one implemented in GMP (which we have not verified). The proof effort is hard to quantify, as the development relies on pre-existing frameworks by the same author for pseudocode and assembly code, but it is rather high. The proofs of the algorithms amount to about 15 000 lines of Coq.

Fischer designed a modular exponentiation library [39] verified using Isabelle/HOL and a framework for verifying imperative programs developed by Schirmer [88]. The verified algorithms include multiplication, division, and square-and-multiply modular exponentiation. The library is not meant to be as efficient as GMP, as it represents arbitrary-precision integers as garbage-collected doubly-linked lists of machine integers. The algorithms are implemented in a restricted variant of the C language and are automatically transcribed into Isabelle. The functional correctness of the algorithms and the pointer-level correctness of the data structure are proved, but not termination or the absence of arithmetic overflows. The author reports running into slowdown and memory issues inside the tool due to the great number of invariants and conditions present in the logical context to keep track of aliasing. By contrast, Why3 automatically keeps track of aliases inside its region-based type system, rather than in the logic. This means that the user does not need to mention in specifications and proofs that such and such pointers are not aliased, which would otherwise cause large slowdown issues similar to those reported by Fischer.

Berghofer developed a verified bignum library programmed in the Spark fragment of the Ada programming language, using a verification framework that sends goals to Isabelle/HOL [10]. The library provides modular exponentiation, as well as the primitives required to implement it: modular multiplication and squaring, modular inverse, and basic operations such as subtraction and doubling. A simple square-and-multiply algorithm is used for modular exponentiation, without the Montgomery reduction or the sliding window optimization that are featured in GMP and WhyMP. The author reports a 150% slowdown compared to OpenSSL for their implementation of RSA using their library. However, OpenSSL uses hand-written assembly code, which accounts for a large part of the discrepancy. The proof effort for the library is only about 2 000 lines of Isabelle written over three weeks, which is surprisingly low, even taking into account the low amount of verified algorithms.

Fixed-point square root algorithms have not been the subject of much formal verification work. However, there has been extensive work on floating-point square root algorithms with quadratic convergence. Harrison used HOL Light to verify such an algorithm for the Intel Itanium architecture [50]. Russinoff used ACL2 to verify the correctness of the AMD K5 floating-point square root [86]. Finally, Rager et al. used ACL2 and interval arithmetic to verify the low-level Verilog descriptions of the floating-point division and square root implementations in the SPARC ISA, and found new optimizations in the process [82].

Bertot et al. verified GMP's square-root general case algorithm [11] using Coq and the Correctness tool, which translates an imperative program and its specifications into verification conditions to be proved with Coq. Our Why3 proof of the same algorithm is directly lifted from their article. They specify the memory as one large array of machine integers, so their specifications must include additional clauses to tell which memory zones are left unchanged. Their formalization is otherwise quite similar to ours. Their proof is about 13 000 lines long, which is about half as long as all our proofs combined. However, Why3 proofs are partially automatic, while Coq proofs are entirely interactive, so it is not surprising that we enjoy a lower proof effort.

Unlike our semi-automatic Why3 proofs, most of the approaches described above use interactive proof assistants. There have also been efforts to prove arithmetic libraries using automated tools. Schoolderman used Why3 to verify hand-optimized Karatsuba multiplication branch-free assembly routines for the AVR microcontroller architecture [89]. The algorithms are not arbitrary-precision, instead there are many routines, each suited for a particular operand size up to 96×96 bits. The fact that the size of the operands is fixed and relatively small means the loops can be unrolled, which is why the algorithms are branch-free. The size being known also makes the proof much easier for SMT solvers, and the authors only needed to add a very small number of annotations to make the automatic proof succeed.

Zinzindohoué et al. developed HACL*, a formally verified cryptography library written in F* and extracted to C [96]. It implements the full NaCl API, and includes a bignum library. The extracted code is as fast as state-of-the-art C implementations, and part of it is now deployed in the Mozilla Firefox web browser. Their approach is similar to ours in that it consists in verifying the algorithms in a high-level language suited for verification, and then compiling them to C. The integers have a small, fixed size that depends on the choice of elliptic curve. Again, the fact that the number sizes are known makes the problem much easier for automated solvers. As a result, their proof enjoys a higher degree of automation than ours. While their specifications are similar or larger in length, the function bodies require much fewer annotations than what we needed for WhyMP.

Finally, Erbsen et al. used Coq to produce Fiat-Crypto, a verified C elliptic curve cryptography library that is faster than existing handwritten implementations [30]. The algorithms are first implemented as high-level, parameter-agnostic templates in continuation passing style. This high-level style facilitates the functional verification of the algorithms. A certified compilation scheme, written in Coq's functional programming language Gallina, specializes these templates for each of the many supported prime fields and produces fast C code that is similar to what an expert implementer may write. Several of these implementations are used in BoringSSL, the SSL library currently used in the Chrome web browser.

4.9.3 WhyMP

In addition to being a test for our verification approach, WhyMP stands on its own as a formally verified C development. It is not quite exhaustive enough to be used as a replacement of GMP outside of carefully chosen contexts, but it could get there with only a moderate amount of work on the `mpz` layer. Its performance is reasonably close to GMP for smaller inputs, in particular when comparing the assembly-free configurations. Enough cosmetic work has been done in the extraction that the generated source code is not much harder to read than GMP's original code. One limitation of WhyMP is a lack of genericity. In the current Why3 development, all number sizes are fixed, we assume that the type `int` is 32 bits wide, and many driver directives rely on GCC/Clang builtins. The library is not as portable as we could expect of a pure C implementation. However, it is not too far off. For example, removing the 128-bit code and using `malloc` instead of `alloca` for temporary allocations is sufficient to make it compile with CompCert. Crucially, the WhyML code itself does not need to be changed.

A natural direction for future work on WhyMP would be to add support for GMP's cryptography-oriented functions. GMP features side-channel resistant variants of most basic operations and modular exponentiation. They are currently not implemented in WhyMP. (Some work was started on the side-channel resistant exponentiation algorithm, but not finished.) As a result, WhyMP cannot really be used in security-sensitive cryptographic applications. GMP also features a number of number-theoretic algorithms, such as an efficient greatest common divisor algorithm and various variants of the Legendre symbol, which WhyMP also lacks.

Another line of future work would be to verify assembly code for some mainstream instruction sets. This way, we could close the performance gap with GMP by also using arithmetic primitives written in handwritten assembly without extending the trusted code base overmuch. In principle, one could develop a memory model and extraction mechanism for x86_64 assembly much as I did for C, and verify assembly primitives that way. One obstacle to this approach is that the control flow of assembly programs is not as structured as that of supported fragment of C. In particular, assembly programs tend to feature backward jumps that may not be supported by Why3's current WP calculus, so we would need to extend it first.

Chapter 5

Proofs by reflection

The previous chapter mentioned that the proof effort required to verify WhyMP's algorithms was large, but did not go into much more detail. Let us now briefly discuss the process of carrying out a Why3 proof in practice. When a user opens a WhyML file with Why3's graphical user interface, they are presented with a number of *tasks*. A task is made up of a logical formula to prove (the goal), and a set of declarations and logical premises (also called the proof context). When the user has managed to prove all the tasks, the WhyML program is considered proved.

Why3 gives the user several tools to prove the tasks. The first resort is the various automated solvers that Why3 interfaces with. The user chooses one or more solvers, sets a time and memory limit, and waits for a solver to declare the goal valid. If that happens, the task is ticked in the graphical interface and the user moves on to the next one. Unfortunately, sometimes no solver manages to prove the goal. This may mean that the goal is in fact invalid. However, there are other reasons why a goal may not be proved by the solvers. For example, the goal may not be well supported by an SMT solver's built-in theories. The user may also have set an overly low time limit. In the case of the verification of WhyMP, these two issues were pervasive. Indeed, many goals featured non-linear arithmetic with symbolic exponents, which many solvers did not handle well. Furthermore, I often set relatively low time limits, such as two to five seconds. Indeed, having a shorter feedback loop tended to make the process more efficient overall, even if it led me to perform extra work for goals that solvers might have eventually proved after several minutes.

Assume now that none of the automated solvers at the user's disposal have managed to discharge a given goal. The user has several remaining options. First, they can attempt to verify the goal in an interactive theorem prover such as Coq or Isabelle. However, I was reluctant to do so, mainly because of my lack of proficiency with these tools. Why3 also offers some native interactive theorem proving capabilities in the form of *task transformations*. Transformations take a task and return one or more tasks that, once proved, imply that the original task is valid. As an example, a transformation might turn the goal `P ∧ Q` into two subgoals `P` and `Q`. The most commonly used transformations are analogous to simple Coq tactics such as `split`, `apply` or `rewrite`. Through sufficient massaging of the goals, the user often succeeds in getting the automated solvers to prove them. However, there is no equivalent to Coq's tactic language for the user to program their own transformations in. They are restricted to Why3's built-in ones.

If the solvers still cannot discharge the goals, the final resort of the user is to edit the original WhyML source file to make it easier to verify. This edit usually takes the form of additional proof annotations such as assertions, lemmas, and so on. When reloading the graphical user interface, the goals change to reflect the new file, and they are hopefully easier to discharge. The user iterates this process until all goals are proved. This iterative process is an instance of what Leino calls *auto-active verification* [62].

During the verification of WhyMP, I had to resort to this method for a very large number of goals. In many cases, the additional proof annotations were themselves difficult to verify, prompting me to add even more. Using the `by` and `so` connectives [20], I was able to write a large number of very long assertions (sometimes more than a hundred lines long!) that looked more like pen-and-paper, declarative proofs than anything else. This explains why the ratio between proof code and program code was so skewed in favor of proofs in the breakdown from Sec. 4.9.

A frustrating fact was that many of these assertions did not contain many clever mathematical insights, but simply performed simple algebraic symbol manipulations that I would have expected the automated solvers to be able to do by themselves. Furthermore, similar kinds of goals ended up being problematic in a variety of WhyMP proofs, and the resulting assertions were themselves very similar to each other. This led me to try to automate these tedious proofs using computational reflection.

This chapter describes my efforts to increase the degree of automation of WhyMP proofs. It is largely drawn from a previous article [70]. I have added to Why3 a framework for proofs by reflection, described in Sec. 5.1. Section 5.2 explains how I used it in the verification of WhyMP. Finally, Sec. 5.3 evaluates the proof effort saved by this increased automation and draws possible lines of future work.

5.1 Introducing reflection in Why3

When one wants to extends a theorem prover with new capabilities (e.g., an inference rule dedicated to the problem at hand), one way is to "incorporate a reflection principle, so that the user can verify within the existing theorem proving infrastructure that the code implementing a new rule is correct, and to add that code to the system" [49]. We have modified Why3 to offer computational reflection, and made use of this to discharge some very large assertions in WhyMP proofs. We first explain the principle of proofs by reflection using the example of Strassen's matrix multiplication algorithm (Sec. 5.1.1). In Sec. 5.1.2, we explain how to reify logical propositions into inductive objects that can be manipulated inside Why3's logic. Section 5.1.3 describes how verified and effectful WhyML programs can be used as decision procedures. Finally, Sec. 5.1.4 discusses the soundness of our approach and its impact on Why3's trusted code base.

5.1.1 An example: Strassen's matrix multiplication

When designing a decision procedure by reflection, one first finds an embedding of the propositions P of interest into the logical language of the formal system. Let us denote $\ulcorner P \urcorner$ the resulting term, e.g., the abstract syntax tree of P. Then one proves that, if $\ulcorner P \urcorner$ satisfies some property φ, then P holds. Thus, when one wants to prove that some proposition P holds, one just has to check that $\varphi(\ulcorner P \urcorner)$ does. If φ is designed so that $\varphi(\ulcorner P \urcorner)$ can be validated just by computations, then we have a proof procedure by computational reflection.

Let us illustrate this process on a toy example: the correctness of Strassen's matrix multiplication algorithm. Among other properties, one has to prove four matrix equalities such as the following one:

$$\mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} = \mathbf{M}_{1,1},$$

with

$$\begin{aligned}\mathbf{M}_{1,1} \quad = \quad & (\mathbf{A}_{1,1} + \mathbf{A}_{2,2}) \cdot (\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) + \mathbf{A}_{2,2} \cdot (\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\ - \quad & (\mathbf{A}_{1,1} + \mathbf{A}_{1,2}) \cdot \mathbf{B}_{2,2} + (\mathbf{A}_{1,2} - \mathbf{A}_{2,2}) \cdot (\mathbf{B}_{2,1} + \mathbf{B}_{2,2}).\end{aligned}$$

By the group laws of matrix addition and by distributivity of matrix multiplication, one easily shows that the right-hand side of the equality can be turned into the left-hand side. Unfortunately, in practice, SMT solvers (Alt-Ergo, CVC4, Z3) and TPTP solvers (the E prover) fail to prove such a proposition. There are two reasons. First, a solver should instantiate the above algebraic laws on the order of one hundred times, assuming they apply them in an optimal way. Second, when verifying programs, the proof context is usually filled with hundreds of other instantiable theorems, which will delay applying the algebraic laws. As a consequence, unless an automated prover implements a dedicated decision procedure for this kind of property, there is no way its proof can be found. Let us see how to supplement the lack of such a dedicated decision procedure.

Embedding terms

The first step is to embed $\mathbf{M}_{1,1}$ into the logical language of Why3. We define the following inductive type `t` to represent its abstract syntax tree:

```
type t = Var int | Add t t | Mul t t | Sub t t | Ext r t
```

Matrices appearing at the leaves of the expression (e.g., $\mathbf{A}_{2,1}$) are assigned a unique integer identifier and are represented using the `Var` constructor. The sum, product, and differences of two matrices, are represented using the constructors `Add`, `Mul`, and `Sub`. Finally, the `Ext` constructor represents the external product (by a value of type `r`), which is not needed in the case of Strassen's algorithm.

Note that the function $\mathbf{M} \mapsto \ulcorner \mathbf{M} \urcorner$ cannot be expressed in the logical language, but its inverse can. We thus define a function that maps a term of type `t` into a matrix, as shown in Fig. 5.1. That definition causes Why3 to create a recursive function `interp` inside the logical system, since its termination is visibly guaranteed by the structural decrease of its argument `x`.

When `aplus`, resp. `atimes`, is instantiated using matrix sum, resp. product, one can prove that the Why3 term `interp (Mul (Add (Var 0) (Var 1))`

```
type vars = int → a
let rec function interp (x: t) (y: vars) : a =
  match x with
  | Var n → y n
  | Add x1 x2 → aplus  (interp x1 y) (interp x2 y)
  | Mul x1 x2 → atimes (interp x1 y) (interp x2 y)
  | Sub x1 x2 → asub   (interp x1 y) (interp x2 y)
  | Ext r x → (#) r (interp x y)
  end
```

Figure 5.1: Interpreting the abstract syntax tree of a polynomial.

`(Var 7))` `y` is equal to $(\mathbf{A}_{1,1} + \mathbf{A}_{1,2}) \cdot \mathbf{B}_{2,2}$, assuming that `y` maps 0 to $\mathbf{A}_{1,1}$, 1 to $\mathbf{A}_{1,2}$, and 7 to $\mathbf{B}_{2,2}$. This proof can be done by unfolding the definition of `interp`, by reducing the `match with` constructs, and by substituting the applications of `y` by the corresponding results. Why3 provides a small rewriting engine that is powerful enough for such a proof, but one could also use an external prover.

Normalizing terms

Let us suppose that we now have two concrete expressions `x1` and `x2` of type `t` and a single map `y` of type `vars`, and that we want to prove the following equality:

```
goal g: interp x1 y = interp x2 y
```

The actual value of `y` does not matter, but the facts that `aplus` is a group operation and that `amult` is distributive do. In other words, we want to see `x1` and `x2` as non-commutative polynomials and we want to prove that they have the same monomials with the same coefficients. To do so, let us turn them into weighted lists of monomials. Figure 5.2 shows an excerpt of the code. For example, the term $(\mathbf{A}_{1,1} + \mathbf{A}_{1,2}) \cdot \mathbf{B}_{2,2}$ gets turned into the list

```
Cons (M 1 (Cons 0 (Cons 7 Nil))) (Cons (M 1 (Cons 1 (Cons 7 Nil))) Nil)
```

```
type m = M int (list int)
type t’ = list m

let rec function interp’ (x: t’) (y: vars) : a =
  match x with
  | Nil → azero
  | Cons (M r m) l → aplus ((#) r (mon m y)) (interp’ l y)
  end

let rec function conv (x:t) : t’
  ensures { forall y. interp x y = interp’ result y }
= match x with
  | Var v → Cons (M rone (Cons v Nil)) Nil
  | Add x1 x2 → (conv x1) ++ (conv x2)
  | Mul x1 x2 → ... (* develop and sort monomials *)
  end
```

Figure 5.2: Converting a polynomial to a list of monomials.

Note that we have introduced a new interpretation function `interp'` and we have stated the postcondition of `conv` accordingly. Why3 requires us to prove that this postcondition holds. The proof is straightforward, even in the multiplication case. Once done, we obtain the following lemma in the context:

```
lemma conv_def: forall x y. interp x y = interp' (conv x) y
```

We define one last function, `norm`, which sorts a weighted list of monomials by insertion using a lexicographic order, merging contiguous monomials along the way. Its postcondition, once proved, leads to

```
lemma norm_def: forall x y. interp' x y = interp' (norm x) y
```

Note that we do not even need to prove that `norm` actually sorts the input list or that it merges monomials, so the proof is again trivial. If there is some bug in `norm`, it only endangers the completeness of the approach, not its soundness, as long as `norm_def` is provable. For example, defining `norm` as the identity function would ultimately be fine but pointless.

By composing `norm` and `conv` and equality, we get our decision procedure φ dedicated to verifying Strassen's algorithm. Indeed, to prove the goal `g` above, we just need to prove the following intermediate lemma:

```
lemma g_aux: norm (conv x1) = norm (conv x2)
```

As with `interp` before, `norm` and `conv` are logic functions defined by induction on their argument, so there is no difficulty in proving `g_aux` using the rewriting engine of Why3 or an external automated prover.

Advantages

There are several advantages to this approach. The most important one is that the user can easily design a decision procedure dedicated to the problem at hand. Indeed, the inductive type for representing expressions does not have to handle the full extent of the language, but can focus on the constructions that matter (e.g., addition). Moreover, the soundness of the system is not endangered, since the user has to prove the correctness of the procedure (e.g., the lemmas `conv_def` and `norm_def`). Finally, since the procedure is ad hoc, performances in the general case do not matter much, so one can write it so that both the code and the proof are straightforward. For instance, in the example above, the sorting algorithm has quadratic complexity and one only has to prove that the interpretation of the list is left unchanged. Thus, SMT solvers quickly discharge all the verification conditions that Why3 generates to guarantee that the implementation of the decision procedure satisfies its specification.

Even if this normalization procedure is dedicated to proving Strassen's algorithm, we took advantage of Why3's module system to make it generic: coefficients are in an arbitrary commutative ring and variables are in a (noncommutative) ring. Both rings are potentially different, as in the case of matrices. The genericity of the presented decision procedure does not extend to supporting variables in a commutative ring, but it is just a matter of duplicating the code of the decision procedure to modify the ordering relation, which we did.

5.1.2 Reification

We have not yet explained how one obtains the inductive objects used to instantiate the decision procedure. Without modifying Why3, it would be up to the user to provide them, which is too much work in practice. Even for an algorithm as simple to verify as Strassen's, the user might forfeit before finishing to translate all the terms of the algorithm.

Possible approaches

To circumvent this issue, the original Why3 proof of Strassen's algorithm uses a clever approach [21]. The type of matrices has been modified so that a matrix contains not only the values of its cells but also the normalized list of monomials representing all the operations performed to obtain the matrix. In other words, the decision procedure has been split and embedded into all the matrix operations and it is executed symbolically along them. The lists of monomials (and the operations to build them) are declared *ghost*, so they do not interfere with actual matrix computations and can be erased from the final algorithm, which is therefore still fundamentally the same. Nonetheless, this approach forces the user to instrument the matrix operations, and while these modifications are suitable to prove Strassen's algorithm, they might be useless when verifying another matrix algorithm, if not detrimental by polluting the proof context with all the symbolic computations.

Thus, for a reflection-based decision procedure to be useful, we have to provide some ways to automate the *reification* process, that is, the conversion of expressions into their inductive representation.

As mentioned above, one difficulty lies in defining $\ulcorner\urcorner$, which is an inverse function of `interp`. This inverse is usually written using the meta-language of a formal system to parse the term and to produce the corresponding inductive object. Since Why3 can load plugins written in OCaml, one could certainly use OCaml as a meta-language for Why3. This unfortunately requires the user to learn the inner workings of Why3.

Another possibility would be to use WhyML as a meta-language by providing some primitives to visit the abstract syntax trees of expressions and by making Why3 able to interpret it. As is the case for other formal systems [95, 29], any WhyML function using such primitives would no longer be meaningful for the remainder of the logical system, so as to avoid inconsistencies. The user would thus no longer need to leave the confines of WhyML, but this is still not completely satisfactory. Indeed, as written before, $\ulcorner\urcorner$ is the inverse of `interp`, so any explicit definition seems superfluous. Instead, I have implemented a Why3 task transformation that can invert such interpretation functions on the fly.

Inversion of the interpretation function

Consider the following function, which is just a variant of the decision procedure for Strassen's algorithm:

```
let norm_f (x1 x2: t) : bool
  ensures { forall y:vars. result = True → interp x1 y = interp x2 y }
= match norm (conv (Sub x1 x2)) with
  | Nil → True (* the difference evaluates to the empty polynomial *)
  | _ → False
  end
```

Whenever the user wants to use this decision procedure to prove a goal, we would like Why3 to automatically find `x1`, `x2`, and `y`, so that the right-hand side of the post-condition matches the goal. This is done by a straightforward recursive walk of the goal. Let us illustrate this walk with `foo a + b = c`. This goal is an equality, and so is the right-hand side of the postcondition of `norm_f`, so the transformation proceeds recursively on each side of the equality. The left-hand side starts with an addition, while there is an application of `interp` in the postcondition, so Why3 assumes that `interp` is an interpretation function.

The `interp` function starts with a pattern matching on its first argument, so Why3 looks at all of the branches. The second branch starts with an addition (i.e., `aplus`, which we assume was instantiated with `+`). So Why3 registers that `x1` should start with the constructor `Add`, and so on, recursively. Eventually, Why3 has to match `foo a` against a branch. None of them matches, but the one for the `Var` constructor returns `y n`, with `y` a variable of type *arrow*. So Why3 selects a fresh integer for `n`, e.g., 0, and remembers that `y` maps 0 to `foo a`.

Extensions

The previous process works fine when a goal has to be proved in isolation, irrespective of the proof context. To remove this limitation, Why3 also recognizes the presence of an implication inside a branch of an interpretation function. In that case, it tries to match a hypothesis of the proof context against the left-hand side of the implication, and it does so recursively until all the hypotheses of the context have been tried. The following functions illustrate this behavior. They serve as interpretation functions of a decision procedure that needs to consider all the equalities from the proof context. In this example taken from the verification of GMP's algorithms, the fact that the goal also has to be an equality is a coincidence.

```
function interp_eq (g:equality) (y:vars) (z:C.cvars) : bool
= match g with (g1, g2) → interp g1 y z = interp g2 y z end

function interp_ctx (l:list equality) (g:equality) (y:vars) (z:C.cvars) : bool
= match l with
  | Nil → interp_eq g y z (* goal *)
  | Cons h t → (interp_eq h y z) → (interp_ctx t g y z)
  end
```

Notice that, since Why3's logical system does not permit functions returning logical propositions, we have defined these interpretation functions as returning Boolean values. But this has no impact on the way reification proceeds.

While the decision procedures presented here ignore quantified formulas, our reification transformation does support them. For example, the excerpt below would handle universal quantifiers in a nameless fashion, using negative indices to store the depth of the quantifier:

```
function interp_fmla (f:fmla) (l:int) (b:vars) : bool
= match f with
  | Forall f' → forall v. interp_fmla f' (l-1) b[l ← v]
  | ...
  end
```

A current limitation of our approach is the purely syntactic nature of the reification step. For example, for an uninterpreted function `foo`, the terms `foo (a+b)` and `foo (b+a)` are mapped to distinct variables, even though they are provably equal. This requires a significant amount of extra work from the user. However, this can be mitigated either in the reification step itself or by composition with another decision procedure.

5.1.3 Effectful decision procedures

Computations in the reflection-based proof from Sec. 5.1.1 are all done in logic functions, which are unfolded by automated provers or Why3's rewriting engine. A limitation of this approach is that Why3's language of logic functions is not very expressive, as they must be side effect-free and their termination must be guaranteed by a structurally decreasing argument.

In this section, we show how we can instead write decision procedures as regular WhyML programs, making full use of the language's imperative features such as loops, references, arrays, and exceptions. These decision procedures are proved correct using Why3 and some automated theorem provers. Their contract can then be instantiated by reification of the goal and context, and used as a cut indication.

Running example: systems of linear equalities

As an example, let us consider a decision procedure for linear equation systems in an arbitrary field (code excerpts in Fig. 5.3). Given some assumed-valid linear equalities in the context, the procedure attempts to prove a linear equality by showing that it is a linear combination of the context.

This is done by representing the context and goal by a matrix and performing a Gaussian elimination (function `gauss_jordan`). In case of success, we obtain a vector of coefficients and we check whether the corresponding linear combination of the context is equal to the goal (function `check_combination`). Otherwise, the procedure returns `False` and proves nothing, since its postcondition has `result = True` as premise.

As is done in Coq with the tactics `lia` and `lra` [12], this is a proof by certificate, since we check if the linear combination of the context returned by `gauss_jordan` matches the goal. There is no need to prove the Gaussian elimination algorithm itself, nor to define a semantics for the matrix passed to it as a parameter. In fact, we do not prove anything about the content of any matrix in the program. This makes the proof of the decision procedure very easy in relation to its length and intricacy.

Let us now examine the contract of the decision procedure. The postcondition states that the goal holds if the procedure returns `True`, for any valuations `y` and `z` of the variables such that the equalities in the context hold. The `valid_ctx` and `valid_eq` preconditions state that the integers used as variable identifiers (second argument of the `Term` constructor) in the context and goal are all nonnegative. This is needed to prove the safety of array accesses. The nature of the reification procedure ensures that these preconditions will always be true in practice, but as reification is not trusted, the user has to verify them explicitly; SMT solvers do this very easily. Finally, the `raises` clause expresses that an exception may escape the procedure (typically an arithmetic error, as we allow the field operations to be partial). In that case, nothing is proven.

Notice that the decision procedure is independent from Why3 (apart from the fact that it is formally verified), in the sense that it does not contain meta-instructions for reification or anything linked to Why3 internals. One could easily imagine finding the same kind of code in an automatic prover.

```
clone LinearEquationsCoeffs as C with type t = coeff

type expr = Term coeff int | Add expr expr | Cst coeff
type equality = (expr, expr)

let linear_decision (l: list equality) (g: equality) : bool
  requires { valid_ctx l }
  requires { valid_eq g }
  ensures { forall y z. result = True → interp_ctx l g y z }
  raises  { C.Unknown → true | Failure → true }
=
  let nv = (max (max_var_e g) (max_var_ctx l)) in
  ...
  let nv = Int63.of_int nv in
  let ll = Int63.of_int (length l) in
  let a = Matrix63.make ll (nv+1) C.czero in
  let b = Array63.make ll C.czero in            (* ax = b *)
  let v = Array63.make (nv+1) C.czero in          (* goal *)
  ...
  fill_ctx l 0;
  let (ex, d) = norm_eq g in
  fill_goal ex;
  let ab = m_append a b in
  let cd = v_append v d in
  match gauss_jordan (m_append (transpose ab) cd) with
    | Some r → check_combination l g (to_list r 0 ll)
    | None → False
  end
```

Figure 5.3: Decision procedure for linear equation systems.

Interpreter

Due to their side effects, functions from WhyML programs only have abstract declarations in the logical world (as opposed to the concrete logic functions used in Sec. 5.1.1). Therefore, they cannot be unfolded by automatic provers or by Why3's rewriting engine. In order to compute the results of decision procedures such as the previous one, I have added an interpreter to Why3. It operates on the ML-like intermediate language produced by the first step of the extraction mechanism (see Sec. 3.4.1). It corresponds to WhyML programs from which logic terms, assertions, and ghost code, were erased, thereby assuming that the program was proved beforehand and that the preconditions are met.

Our interpreter provides built-in implementations for some axiomatized parts of the Why3 standard library, such as integer arithmetic, arrays, and references. To ease the debugging of decision procedures, I have added to Why3's standard library a `print` function of type `'a -> unit` that does nothing. It is interpreted as a polymorphic `printf` function.

5.1.4 Soundness, trusted code base

The implementation of our framework requires two additions to Why3: a reification transformation and an interpreter of WhyML programs. Let us discuss the soundness of our approach.

First, the rather large and intricate code needed for reification is not part of the trusted computing base of Why3. Indeed, the reification merely guesses values for all the relevant variables and asks Why3 to instantiate the contract of the decision procedure with them. Assuming the user has proved the soundness of the decision procedure, this instantiated proposition holds, whether the reification algorithm is correct or not. A reification failure would either prevent a well-typed instantiation of the post-condition, or the resulting cut would be useless for proving the current goal.

Contrarily to the reification code, our interpreter is part of the trusted computing base. Fortunately, it is quite simple, since it only manipulates concrete values. There is no need for partial evaluation nor symbolic execution nor polymorphic equality, which makes this new interpreter much simpler than the existing rewriting engine. Another reason for its simplicity is that the interpreted language is not WhyML, but the intermediary representation used by the extraction mechanism (see Sec. 3.4.1). This intermediate language has relatively few constructions, since program transformations performed by the existing extraction mechanism eliminate potentially confusing behaviors from the surface language such as parallel assignation.

5.2 Proofs by reflection in WhyMP

The motivation that led me to develop this framework for proofs by reflection was the large amount of tedious proofs that had to be performed during the early stages of the verification of WhyMP. This section explains how I was able to use computational reflection to automate some of these proofs. We begin by presenting a tedious proof from WhyMP as a motivating example (Sec. 5.2.1). Then, we reduce the problem to a system of equations that can be seen as linear using a well-chosen set of coefficients (Sec. 5.2.2). Finally, Sec. 5.2.3 shows how the decision procedure from the previous chapter can be composed with some well-chosen normalization functions to solve the problem.

5.2.1 Motivating example: an easy yet tedious proof

Many proofs in WhyMP involve very large assertions that use the by and so connectives. Many of these assertions are proofs of relatively simple identities through symbolic manipulation of algebraic expressions. Their proofs are straightforward, but time-consuming when there are many intermediate steps. In an ideal world, automated provers would be able to discharge these by themselves, but they are not. An excerpt of an earlier version of the WhyMP long addition function can be found in Fig. 5.4. It features one such assertion.

```
let wmpn_add_n (r x y: ptr uint64) (sz: int32) : uint64
  requires { valid x sz ∧ valid y sz ∧ valid r sz }
  ensures { 0 ≤ result ≤ 1 }
  ensures { value r sz + (power radix sz) * result = value x sz + value y sz }
=
  let ref i = 0 in
  let ref lx = 0 in
  let ref ly = 0 in
  let ref c = 0 in
  while Int32.(<) i sz do
    invariant { value r i + (power radix i) * c = value x i + value y i }
    lx ← get_ofs x i;
    ly ← get_ofs y i;
    let res, carry = add_with_carry lx ly c in
    set_ofs r i res;
    c ← carry;
    assert { value r (i+1) + (power radix (i+1)) * c
             = value x (i+1) + value y (i+1)
             by value r (i+1) + (power radix (i+1)) * c
               = value r i + (power radix i) * res
                 + (power radix i) * c
               = ... (* 10+ subgoals *) };
    i ← i + 1;
  done;
  c
```

Figure 5.4: WhyML proof of the long addition function.

The proof of this assertion consists in a sequence of about ten rather simple steps (rewrite an equality in the context, use distributivity, etc.) but the large search space prevents the automatic provers from succeeding. Therefore, we had to provide many cut indications by hand using the by construct.

Yet, with a judicious choice of coefficients, this goal (and many others in the proofs of our library) can be seen as a linear combination of the context. Therefore, we should be able to use the decision procedure from Sec. 5.1.3 to prove the assertion in one go. In the up-to-date version, only the goal is stated, and there is no need for a by keyword and a subsequent proof. Let us now explain how.

5.2.2 Coefficients

The following is a simplified version of the context and goal obtained for the assertion of the main loop of wmpn_add_n (Fig. 5.4, line 17). The variables r1 and c1 denote the values of r and c at the start of the loop (before the modifications that occur at lines 15 and 16). Notice that the linear combination $\mathtt{H5} - \mathtt{H4} - \mathtt{H3} + \mathtt{H2} + \beta^{\mathtt{i}} \cdot \mathtt{H1} + \mathtt{H}$ simplifies to an equality equivalent to g. In order to prove this, our decision procedure has to include powers of β (radix in the WhyML code) in its coefficients, and to support symbolic exponents (as i is a variable).

```
axiom H:  value r1 i + (power radix i) * c1 = value x i + value y i
axiom H1: res + radix * c = lx + ly + c1
axiom H2: value r i = value r1 i
axiom H3: value x (i+1) = value x i + (power radix i) * lx
axiom H4: value y (i+1) = value y i + (power radix i) * ly
axiom H5: value r (i+1) = value r i + (power radix i) * res
goal g:   value r (i+1) + power radix (i+1) * c
          = value x (i+1) + value y (i+1)
```

More precisely, the coefficients of our decision procedure are the product of a rational number and a (symbolic) power of β. Fig. 5.5 is an excerpt of the WhyML implementation of the coefficients. The decision procedure of Fig. 5.3 is instantiated with type coeff = t.

```
type exp = Lit int | Var int | Plus exp exp | Minus exp | Sub exp exp
type rat = (int, int)
type t = (rat, exp)

function qinterp (q:rat) : real
= match q with (n,d) → from_int n /. from_int d end

function interp_exp (e:exp) (y:vars) : int
= match e with
  | Lit n → n
  | Var v → y v
  | Plus e1 e2 → interp_exp e1 y + interp_exp e2 y
  | Sub e1 e2 → interp_exp e1 y - interp_exp e2 y
  | Minus e’ → - (interp_exp e’ y)
  end

function interp (t:t) (y:vars) : real
= match t with
  (q,e) → qinterp q *. pow radix (from_int (interp_exp e y))
  end
```

Figure 5.5: Definition of the coefficients.

One can define addition, multiplication, and multiplicative inverse over these coefficients. Addition is partial, since one may only add two coefficients with equal exponents (otherwise the result would not be a valid coefficient). If this is not the case, the addition raises an exception, which is accounted for in the specification of the decision procedure (exception C.Unknown in Figure 5.3). Note that exponents do not have to be structurally equal, only to have equal interp_exp interpretations for all values of y, which can be automatically proved within the decision procedure.

5.2.3 Modular decision procedures

The coefficients above are expressive enough to prove assertions such as the one in Figure 5.4. However, notice that their interpretation (function `interp` in Figure 5.5) is expressed in terms of real numbers (this is needed because the Gaussian elimination algorithm used in the decision procedure needs to compute the multiplicative inverse of some coefficients), while the context and goal consist in equalities over integers. Moreover, the inductive type for expressions that is used in the decision procedure (type `expr` in Figure 5.3) is quite restrictive, which simplifies the code of the decision procedure. However, this is problematic for the user, since a term such as `2 * 3 * x` cannot be reified by inversion of `interp`.

These constraints can be lifted thanks to an approach similar to the `conv` function in Sec. 5.1.1. We compose the decision procedure `linear_decision` with a function that converts integer-valued coefficients to real-valued coefficients (called by `m_ctx` and `m_eq`), and the function `simp_ctx`, which converts from a more expressive expression type to the `expr` type (code excerpts in Figure 5.6).

```
let decision (l:list equality') (g:equality') : bool
  requires { valid_ctx' l ∧ valid_eq' g }
  ensures { forall y z. result = True → interp_ctx' l g y z }
  raises  { Unknown → true }
= let sl, sg = simp_ctx l g in
  linear_decision sl sg

let mp_decision (l: list equality'') (g: equality'') : bool
  requires { valid_ctx'' l ∧ valid_eq'' g }
  ensures  { forall y z. result = True → pos_ctx'' l z → pos_eq'' g z
             → interp_ctx'' l g y z }
  raises  { Unknown → true }
= decision (m_ctx l) (m_eq g)
```

Figure 5.6: Composition of decision procedures.

The conversion procedure from integer-valued to real-valued coefficients is only sound when the exponents of β are nonnegative. This is always the case for GMP algorithms. Due to the symbolic exponents, it is not yet possible to automatically prove this property within the decision procedure, so we instead add it as an extra precondition (the `pos_*` predicates in `mp_decision`). In practice, SMT solvers prove it easily.

While the final decision procedure is specialized for WhyMP goals, almost all the reasoning is done in the generic linear decision procedure `linear_decision`, presented in the previous section. For other use cases than GMP, users should also be able to develop their own interpretation and conversion layers and reuse the primary linear decision procedure (available in the source code of WhyMP) as is.

5.3 Evaluation, perspectives

Let us now compare this reflection framework to previous work, evaluate how useful it was to the verification of WhyMP, and lay out future work perspectives.

5.3.1 Related work

Computational reflection is a widely used way to automate proofs in formal verification systems. A few examples are discussed below.

Grégoire and Mahboubi developed a reflexive tactic to efficiently decide equality over rings and semi-rings in the Coq theorem prover [48]. The core decision procedure normalizes polynomials of $C[X]$ into sparse Horner form, where the set C of coefficients is a parameter. The algorithm is optimized and the normal form is carefully chosen to maximize performance. However, a large part of the computations occurs in the underlying coefficient set, which is generally a Coq type that is not necessarily computationally efficient. For example, the authors report that the most commonly used coefficient set is `Z`, a representation of relative integers using lists of bits. Reification is performed by relatively simple functions written in Coq's meta-language Ltac, which allows pattern matching over Coq terms. Presumably, a user wishing to use the tactic on an unusual ring would need to write their own reification function in Ltac, but most veteran Coq users can be expected to be able to do so.

Besson developed the Coq tactics `lia` and `lra` (named after the Linear Integer Arithmetic and Linear Real Arithmetic SMT theories) using a different approach [12]. Rather than programming a decision procedure inside Coq, a skeptical approach is used. The proof search is delegated to external solvers. They produce certificates that are then checked by a verified decision procedure implemented in Coq.

There is not as much existing work on computational reflection using effectful decision procedures. One may cite Claret et al. [19]. They use a monadic encoding of effectful computations in Coq (e.g., non-termination). Monadic decision procedures are turned into impure programs that are executed outside of Coq. The result of these external computations is used as a "prophecy" to simulate the execution of the decision procedure inside of Coq. Since we are working with Why3, which natively supports impure computations, we sidestep the need for a heavyweight simulation mechanism.

Chaieb and Nipkow implemented quantifier elimination procedures for both linear integer arithmetic and linear real arithmetic in Isabelle/HOL [18]. They implemented both procedures twice: once directly as Isabelle tactics written in a meta-language, and once by reflection, that is, as verified programs that perform computations inside the logic, and can be extracted to ML code using Isabelle's code generator. They benchmarked both implementations and report that the reflexive approach is one to two orders of magnitude faster. The key advantage seems to be that the execution environment of the tactics is much slower than native extracted ML code.

Martínez et al. introduced a metaprogramming framework in F* that allows the development of reflexive decision procedures [68]. The framework converts back and forth between F* terms and an embedding of their abstract syntax trees, thus enabling reification. Using this framework, the authors developed a `ring`-like reflexive decision procedure to automate the resolution of non-linear arithmetic goals and increase the replay speed and robustness of their proof of a real-world cryptographic algorithm. Using F*'s extensible effect system, they define metaprograms as first-class F* programs that have a particular effect: manipulating the proof state. Metaprograms can be run using F*'s reduction engine, or extracted to OCaml and run natively. They can be specified and verified, but only to a degree. Safety properties can be verified, but the fact that a tactic correctly solves a goal cannot be proven, or even expressed in F*'s logic. By contrast, our framework lets users verify that their decision procedures are correct, since the reification transformation is kept separate from the decision procedures.

5.3.2 WhyMP and proofs by reflection

When I developed this reflection framework at the end of 2017, WhyMP only contained a few algorithms: addition, subtraction, schoolbook multiplication, logical shifts, and division. At that point, the library contained about 6000 lines of WhyML code, including 4000 lines of proof code. After completing the decision procedure for linear equation systems and its specialization for GMP goals, I revisited the existing proofs to see if many assertions could be replaced by proofs by reflection.

It was reasonably successful. I was able to delete almost a thousand lines' worth of assertions, including about half of the proof code of the addition, subtraction and multiplication proofs. The division contained a much bigger amount of assertions, many of which were not in the fragment supported by my decision procedure. It is worth noting that while it only deals with linear equation systems, I was able to use it to prove goals in the proofs of multiplication, division, and logical shifts that appeared completely nonlinear at first glance. Indeed, in many cases, it was possible to replace fifty-line long assertions by only one or two short proof indications containing the nonlinear part of the proof, and let the decision procedure do the rest. However, careful manual review was necessary to figure out which indications should be added.

This brings us to the main limitation of the reflection framework in its current state. From the user's perspective, the reflection framework is implemented as a Why3 transformation. The user invokes the transformation from the Why3 IDE by supplying the name of the decision procedure to be used. Why3 attempts to reify the goal, infers the parameters to pass to the decision procedure and executes it. If it is successful, the original task is replaced by a set of simple tasks that check that the decision procedure's postcondition implies the original goal, and check the decision procedure's preconditions. However, if the decision procedure does not succeed in finding a proof, the user is left with this information and not much actionable feedback. (Indeed, our "decision procedures" are really semi-decision procedures.) I made liberal use of the `printf` function mentioned in Sec. 5.1.3 in the process of developing the linear equations decision procedure, which helped a little, but the problem largely remains. In the end, it is still difficult for the user to understand what should be changed to help a proof by reflection succeed.

Unfortunately, I did not end up using the reflection framework all that much in subsequent proofs. While the existing decision procedure was used a few times, I never found a good case for implementing new ones. Indeed, developing and proving a usable decision procedure has a non-trivial cost. In order to justify this cost, a large number of sufficiently similar goals must exist. Apart from the linear equation systems, the only other type of goal in the WhyMP proof that may justify this is the instantiation of `mpz` frame lemmas. Indeed, since my `mpz` memory model does not use Why3's alias tracking (see Sec. 4.6), a large amount of proof effort in the `mpz` proofs was spent convincing Why3 that the state of most `mpz` variables was left unchanged by various operations. However, by the time I realized how recurring this issue was, most of the proofs were already completed, so it was not really worth going back and developing a dedicated decision procedure for it.

5.3.3 Perspectives

The framework for proofs by reflection is not in a very mature state yet. It is available in Why3, but hasn't been used much outside of WhyMP. Indeed, it is difficult to use compared to the SMT solvers, and the reification transformation does not cover quite as many cases as I would like. Getting it to work on WhyMP proofs was a rather tedious process. Still, automating proofs that used to require hundred-line long assertions was a promising success. In order to make this framework into a more mature tool, an important priority is to find a way to provide more feedback to the user, so that they can understand more easily why their reflection proofs do not succeed.

The performance of the decision procedures has not been a big issue so far. The most difficult goal that was solved by the linear equations decision procedure was an assertion in the proof of the schoolbook division that required performing Gaussian elimination on a matrix of size about 150×90. The decision procedure terminated in about 3 seconds, which was acceptable from a user-experience standpoint. If larger goals become problematic, one possible solution to improve performance would be to give up on our WhyML interpreter, extract the decision procedures all the way to OCaml and execute the resulting binaries.

Finally, a more ambitious project could be to partially automate the user's role in Why3 proofs. Indeed, my experience when verifying WhyMP has been that similar user actions were often needed to prove syntactically similar goals. For examples, goals that were conjunctive formulas very often required the use of a split transformation. Goals that involved a certain predicate almost always needed to be inlined. Goals that required non-linear arithmetic could often only be solved by a specific automated prover, and goals that required instantiating many lemmas by another. In the end, for many simple goals I only needed to perform a brief syntactical analysis of the goal before choosing the correct action in the graphical user interface. However, the sheer amount of such goals made it a very time-consuming activity. I would have saved a large amount of time if I had been able to program a decision procedure to attempt to make some of these decisions automatically. The possibility of extending the current framework by embedding a representation of user actions in a built-in theory, essentially enabling users to implement their own tactics as WhyML programs, seems worth exploring.

Chapter 6

Conclusion

This thesis makes three contributions. I have verified a clone of an extensive and efficient arbitrary-precision arithmetic library using Why3. In order to be able to do so, I have added to Why3 the tools needed to verify and extract C programs. Finally, in order to increase the degree of automation of my verification work, I have developed a framework for proofs by reflection in Why3. My technical choices were already evaluated in the conclusions of each chapter, so we simply summarize each of these contributions briefly. We then go over a few longer-term challenges, and discuss possible lines of future work.

6.1 Contributions

WhyMP. I have used Why3 to verify a significant subset of the algorithms in the `mpn` layer of the GMP library (using Mini-GMP for the base conversion functions), as well as a smaller fragment of the `mpz` layer. The result is a verified arbitrary-precision library called WhyMP. It is compatible with GMP in terms of number representation and function signatures. It also preserves the vast majority of the implementation tricks found in GMP's source code. As a result, it is competitive with the assembly-free version of GMP in terms of performance. The regular version of GMP, which uses assembly routines for the most critical inner loops, is twice as fast. However, by incorporating a small selection of (unverified) multiply-and-accumulate assembly primitives from GMP into the trusted code base, WhyMP can be brought closer to GMP, up to a 5-20% slowdown depending on the operation.

WhyMP was a huge undertaking. It took more than twenty thousand lines of verified WhyML code, which were written over four years. It goes far beyond the existing verified arbitrary-precision arithmetic libraries in terms of amount and intricacy of algorithms, as well as performance. The extracted C code is available at `https://gitlab.inria.fr/why3/whymp/`. The WhyML code and proofs can be found in the `examples/multiprecision` directory packaged with the source code of Why3 1.3, available at `https://gitlab.inria.fr/why3/why3/`.

Verification of C programs with Why3. In order to make the development of WhyMP possible, I have extended Why3's extraction mechanism to accept C as a target language, and added to Why3's standard library a model of C's memory and datatypes. As a result, Why3 users may now write WhyML programs on top of these models, and these programs can be compiled to idiomatic C code. These additions to Why3 enable a method for developing verified C programs that compares favorably with the state of the art. It scales up to the verification of multiple thousands of lines worth of C code, while also enabling the user to verify very complex goals by leveraging powerful theorem proving capabilities.

The verification of WhyMP informed the development of the C extraction mechanism and memory models in many ways that toy examples could not have, discovering both subtle bugs and missing features. While the C extraction mechanism is still not quite as mature as I would like, I expect it to steadily improve as it sees more use.

In the case of small programs that do not make use of any C-specific features, the embedding of C into WhyML comes almost for free, in that it does not make the programs significantly more difficult to verify. As a result, other Why3 users have started to use the C extraction mechanism in their own work, even in the situations where using C would not have been strictly required [38].

C is the first imperative language with manual memory management to be supported by Why3 in this fashion, and it paves the way to new ones. Indeed, the approach I have presented is not specific to C in nature. Memory models and extraction printers for other languages can be developed as needed. Moreover, part of the work done on the readability of the extracted code (precedence system, proxy variable elimination) is independent of the target language. Indeed, the readability of the OCaml code produced by Why3's extraction mechanism has sharply improved as a side effect of this work.

Proofs by reflection. In the early stages of the verification of WhyMP, many seemingly easy goals required a surprisingly high amount of proof effort in the form of very long proof annotations. In an effort to increase the degree of automation of these proofs, I have added to Why3 a framework for proofs by reflection. Within this framework, I was able to implement a dedicated decision procedure as a formally verified program. Interestingly, such decision procedures make full use of WhyML's imperative features, whereas existing frameworks for reflection proofs require decision procedures to have no side effects. Using my decision procedure, I was able to replace almost a thousand lines of proof annotations by automated proofs.

6.2 Challenges and roadblocks

Scalability and proof robustness. In addition to the tools specific to the verification of C programs, WhyMP also had some influence on the development of Why3 at large. WhyMP is arguably the largest Why3 development in total code length. It breaks records by far on a number of metrics, such as the number of subgoals in a single function, the number of subgoals in a single assertion, or the number of subgoals in a proof session. As a result, WhyMP exercises the scalability of Why3 in many interesting ways. For example, the Toom-Cook multiplication algorithm is implemented as a set of mutually recursive functions. Since they are mutually recursive, they need to be in the same WhyML source file. As a result, this source file is several thousand lines long. At some points, everytime a change was made to the file, it took Why3 almost two minutes to parse the file and match each subgoal to their existing proofs. This issue has now been fixed by Why3's developers.

Proof robustness was another recurrent issue. When a WhyML source file is modified, the generated verification conditions may not be the same as the previous ones. Why3 attempts to preserve as much of the proof by pairing the new goals with the old ones, but this is not always successful, especially for sessions where many transformations were applied through the graphical user interface. Even when pairing is successful, the proof context may change in ways that make automated provers fail at replaying the proofs. As a result, modifying a file that is high up in the dependency tree requires spending a nontrivial amount of effort getting old proofs to replay, which puts a significant damper on experimentation. This is now also partly fixed. Work on increasing the robustness of Why3 proofs is ongoing and is a necessity as the tool matures.

Trusted code base. In order to be convinced that WhyMP is correct, one needs to trust several components. Some formal verification systems such as Coq or Isabelle are built in such a way that only a small kernel of code needs to be trusted. This is not the case for Why3. The component that performs the WP computations and produces verification conditions is implicitly trusted, as well as the typer (especially the region-based alias handling), and the various goal transformations that can be applied from the user interface. Fortunately, the verification condition generator relies on well-established theoretical principles and has been used in numerous occasions by now. While its codebase is larger than ideal, it would be plausible to trust it. There is also ongoing work on making Why3 task transformations output a machine-checkable certificate [43], with the eventual goal of removing them from the trusted code base.

One also needs to trust that the goals are accurately translated by Why3 into the input languages of the various automated solvers, as well as in the correctness of the solvers themselves. Most automated solvers used in the proof are off-the-shelf tools and have been used in a variety of contexts. While this does not ensure that they are free of bugs, they are not the most worrying component. It is also possible to increase the confidence in a given proof by systematically requiring several provers to agree on a goal before considering it proved, although this was not done for WhyMP.

Various underlying WhyML theories (memory model, arithmetic primitives, lemma libraries, etc.) should also be trusted. There is no real way around the fact that trusted axioms exists. The main line of defence against errors in these axioms is to make the trusted theories as short as possible, so that they may be reviewed manually.

Trusting these various components, one can be confident that WhyMP as a WhyML program is correct. The manual translation from the GMP source code to WhyML is unverified, which means that WhyMP may not be a precise clone of GMP, although that does not affect the correctness of WhyMP as a standalone program. More importantly, the extraction procedure from WhyML to C, as well as the associated extraction driver, should also to be trusted in order to be convinced of the correctness of WhyMP as a C library. This component is more worrying. It is a relatively recent addition to Why3, and the C printer even more so. Although the development of WhyMP put it to the test and uncovered a significant amount of bugs, it is not yet as mature as one would want it to be. The principles of the extraction mechanism have been the subject of a pen-and-paper formalization [79], but this is not the case of the C printer, and the existing formalization does not take the extraction drivers into account. A natural line of future work would be to strive to increase the trust in the C extraction with a proof that it preserves the semantics of the extracted program. The first step would probably be the development of a formalized semantics of WhyML.

Automation and expertise. The reason why WhyMP took so much effort to verify is not fully represented by the amount of lines of verified code. Indeed, a significant part of the proof effort for the verification of WhyMP was spent trying to understand why the algorithms were correct. This would have posed no issues to an expert, but arbitrary-precision arithmetic was initially far from my specialty. More generally, the formal verification of non-trivial programs tends to require both domain-specific expertise and expertise in formal methods. This is a significant obstacle to the spread of formally verified software. It seems to me that an important long-term goal of formal verification research should be to develop tools and methods that are accessible to users who are not experts in formal methods. In an ideal world, developers of critical code should be able to verify their software even as it is being developed.

Looking back to the verification of WhyMP, the only step that came close to this ideal in terms of automation was the verification of the first half of the `mpn_sqrt1` function, that is, the Newton iteration that computes the square root of a machine integer. In order to perform this proof, we had to develop a WhyML model of fixed-point arithmetic, but it can now be reused in future proofs. We also had to write four simple assertions that connected the error terms of the various steps of the Newton iteration. However, once this was done, the Gappa tool was able to automatically prove the error bounds at the end of the iteration, which was the most difficult part of the proof by far. It would have taken me much more time to write a pen-and-paper proof, or even to fully convince myself that the algorithm was correct, than it took me to formally verify it using Why3 and Gappa. The realization that I had formally verified an algorithm that I did not fully understand was one of my happiest moments of this work. This sort of automation is also what I believe is required, in order for non-experts to use our tools.

The key factor to this success was that the goal, while complex, fell squarely within the domain of the Gappa solver. More precisely, it was easy to isolate the parts of the goal that were not and treat them separately (the four aforementioned assertions), and let the tool do the rest of the work. This is evidence in favor of Why3's strategy of interfacing with a wide collection of automated solvers. The built-in theories of real number arithmetic featured in the mainstream general-purpose SMT solvers would not have been able to discharge these goals, but a specialized tool did the trick. But what to do when we have another very specialized problem and no off-the-shelf tool suited to tackle it? A possibility is to develop a verified dedicated decision procedure for it within our new framework for proofs by reflection in Why3, as we did to solve the almost-linear-arithmetic goals from last chapter.

6.3 Future work

WhyMP. WhyMP is not quite a suitable replacement for GMP yet. The main reason is a lack of exhaustivity in the `mpz` routines, although that can likely be fixed in a few months of work for those that rely on already implemented `mpn` routines. WhyMP also lacks GMP's number theory functions. I expect their verification to be a more ambitious project which will likely only be undertaken if a specific need arises. Finally, WhyMP does not implement GMP's cryptography-oriented functions. In order to do so in an interesting way, I would like to first find a way to specify and prove their side-channel resistance properties, that is, the fact that the control flow and execution times do not depend on the input. Another ambitious continuation to this project would be to develop the tools needed to verify assembly code in Why3, allowing us to bring WhyMP to the level of performance of GMP's main configuration without needing to trust extra code.

Proofs by reflection. Although I replaced many assertions by automated proofs using my initial decision procedure, I did not end up using the reflection framework very much in subsequent WhyMP proofs. One reason is that these proofs did not involve a large amount of goals that were well covered by my decision procedure, and I did not find another large set of similar goals that justified implementing a new one. Another reason is that the reflection framework is not a very mature tool yet, and I prioritized completing WhyMP proofs over improving it. In the medium term, it seems important to find a good way to provide actionable feedback to the users that helps them understand why their proofs by reflection do not succeed.

In the longer term, an enticing prospect could be to enable users to automate transformation and prover calls in their decision procedures, essentially extending the reflection framework into a more general tactic system. Ideally, such a system would be able to automate the separation of concerns between subgoals that can be discharged by an automated prover or decision procedure, and those that require user intervention.

Bibliography

[1] *The ASCII Graphic Character Set*, 1975.

[2] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[3] Jean-Raymond Abrial. *The B-Book, assigning programs to meaning*. Cambridge University Press, 1996.

[4] Reynald Affeldt. On construction of a library of formally verified low-level arithmetic functions. *Innovations in Systems and Software Engineering*, 9(2):59–77, 2013.

[5] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. Cogent: Verifying high-assurance file system implementations. *ACM SIGARCH Computer Architecture News*, 44(2):175–188, 2016.

[6] Andrew W. Appel. Verified software toolchain. In *European Symposium on Programming*, pages 1–17. Springer, 2011.

[7] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye. *WP Plug-in Manual*, 2020.

[8] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI C specification language, 2008.

[9] Josh Berdine, Cristiano Calcagno, and Peter W. O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[10] Stefan Berghofer. Verification of dependable software using SPARK and Isabelle. In Jörg Brauer, Marco Roveri, and Hendrik Tews, editors, *6th International Workshop on Systems Software Verification*, volume 24 of *OpenAccess Series in Informatics (OASIcs)*, pages 15–31, Dagstuhl, Germany, 2012.

[11] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3-4):225–252, 2002.

[12] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In Thorsten Altenkirch and Conor McBride, editors, *International Workshop on Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 48–62, Nottingham, UK, 2007.

[13] Régis William Blanc, Etienne Kneuss, Viktor Kuncak, and Philippe Suter. An overview of the Leon verification system: Verification by translation to recursive functions. In *4th Annual Scala Workshop*, 2013.

[14] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. Ghosts for lists: A critical module of Contiki verified in Frama-C. In *NASA Formal Methods Symposium*, pages 37–53. Springer, 2018.

[15] Marco Bodrato and Alberto Zanoni. Integer and polynomial multiplication: Towards optimal Toom-Cook matrices. In *2007 International Symposium on Symbolic and Algebraic Computation*, pages 17–24. ACM, 2007.

[16] Alfred Brauer. On addition chains. *Bulletin of the American mathematical Society*, 45(10):736–739, 1939.

[17] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic.* Cambridge University Press, 2010.

[18] Amine Chaieb and Tobias Nipkow. Proof synthesis and reflection for linear arithmetic. *Journal of Automated Reasoning*, 41(1):33–59, 2008.

[19] Guillaume Claret, Lourdes del Carmen González Huesca, Yann Régis-Gianas, and Beta Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *4th International Conference on Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 67–83. Springer, July 2013.

[20] Martin Clochard. Preuves taillées en biseau. In *vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA)*, Gourette, France, January 2017.

[21] Martin Clochard, Léon Gondelman, and Mário Pereira. The Matrix reproved. *Journal of Automated Reasoning*, 60(3):365–383, 2017.

[22] Cyrille Comar, Johannes Kanig, Yannick Moy, Johannes Kanig, Rod Chapman, Cyrille Comar, Jerôme Guitton, Yannick Moy, Emyr Rees, Sylvain Conchon, et al. Why Hi-Lite Ada? In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, volume 198, pages 51–69. Springer, 2011.

[23] Stephen A. Cook. *On the minimum computation time of functions.* PhD thesis, Department of Mathematics, Harvard University, 1966.

[24] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005.

[25] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *10th International Conference on Software Engineering and Formal Methods*, number 7504 in Lecture Notes in Computer Science, pages 233–247, 2012.

[26] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1):1–20, 2010.

[27] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2010.

[28] Edsger W. Dijkstra. *A discipline of programming*, volume 1. Prentice-Hall Englewood Cliffs, 1976.

[29] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. In *22nd ACM SIGPLAN International Conference on Functional Programming*, pages 34:1–34:29, Oxford, UK, September 2017.

[30] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219. IEEE, 2019.

[31] Federal Aviation Administration. Airworthiness directives, pages 24789–24791, 2015.

[32] Gaspard Férey and Natarajan Shankar. Code generation using a formal model of reference counting. In *NASA Formal Methods Symposium*, pages 150–165. Springer, 2016.

[33] Jean-Christophe Filliâtre. Verifying two lines of C with Why3: an exercise in program verification. In *International Conference on Verified Software: Tools, Theories, Experiments*, pages 83–97. Springer, 2012.

[34] Jean-Christophe Filliâtre. One logic to use them all. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Lake Placid, USA, June 2013.

[35] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016.

[36] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007.

[37] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128, Heidelberg, Germany, March 2013.

[38] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and Genericity in Why3. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, October 2020.

[39] Sabine Fischer. Formal verification of a big integer library. In *DATE Workshop on Dependable Software Systems*, 2008.

[40] Robert W. Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[41] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2):13–es, 2007.

[42] Clément Fumex, Claire Dross, Jens Gerlach, and Claude Marché. Specification and proof of high-level functional properties of bit-level programs. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *8th NASA Formal Methods Symposium*, volume 9690 of *Lecture Notes in Computer Science*, pages 291–306, Minneapolis, MN, USA, June 2016.

[43] Quentin Garchery, Chantal Keller, Claude Marché, and Andrei Paskevich. Des transformations logiques passent leur certificat. In *JFLA 2020 - Journées Francophones des Langages Applicatifs*, Gruissan, France, January 2020.

[44] Léon Gondelman. *A Pragmatic Type System for Deductive Software Verification.* PhD thesis, Université Paris-Saclay, December 2016.

[45] Torbjörn Granlund and the GMP development team. GNU MP: The GNU multiple precision arithmetic library, 6.1.2. 2016.

[46] David Greenaway. *Automated proof-producing abstraction of C code.* PhD thesis, University of New South Wales, Sydney, Australia, 2014.

[47] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: formal verification of C code without the pain. *ACM SIGPLAN Notices*, 49(6):429–439, 2014.

[48] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Tom Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, pages 98–113, Oxford, UK, August 2005.

[49] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre, 1995.

[50] John Harrison. Formal verification of square root algorithms. *Formal Methods in System Design*, 22(2):143–153, 2003.

[51] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[52] Thierry Hubert and Claude Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 190–199. IEEE, 2005.

[53] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages – C*, 2000.

[54] Bart Jacobs and Frank Piessens. The VeriFast program verifier. *CW Reports*, 2008.

[55] Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet Physics Doklady*, volume 7, pages 595–596, 1963.

[56] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[57] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[58] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In Tiziana Margaria and Bernhard Steffen, editors, *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 9952 of *Lecture Notes in Computer Science*, pages 461–478, Corfu, Greece, October 2016.

[59] Herb Krasner. The cost of poor software quality in the US: A 2018 report, 2018.

[60] Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[61] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[62] K. Rustan M. Leino and Michał Moskal. Usable auto-active verification. In *Usable Verification Workshop*, Redmond, WA, USA, November 2010.

[63] Xavier Leroy et al. The CompCert verified compiler, 2012.

[64] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[65] Jacques-Louis Lions. Ariane 501 failure: Report by the inquiry board. 1996.

[66] William Mansky, Andrew W. Appel, and Aleksey Nogin. A verified messaging system. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–28. ACM New York, NY, USA, 2017.

[67] Claude Marché and Yannick Moy. The Jessie plugin for deductive verification in Frama-C. *INRIA Saclay Île-de-France and LRI, CNRS UMR*, 2012.

[68] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hrițcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, et al. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *European Symposium on Programming*, pages 30–59. Springer, 2019.

[69] Steve McConnell. *Code complete.* Pearson Education, 2004.

[70] Guillaume Melquiond and Raphaël Rieu-Helft. A Why3 framework for reflection proofs and its application to GMP's algorithms. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *9th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Computer Science, pages 178–193, Oxford, United Kingdom, July 2018.

[71] Guillaume Melquiond and Raphaël Rieu-Helft. Formal verification of a state-of-the-art integer square root. In *IEEE 26th Symposium on Computer Arithmetic (ARITH)*, Kyoto, Japan, June 2019.

[72] Guillaume Melquiond and Raphaël Rieu-Helft. WhyMP, a formally verified arbitrary-precision integer library. In *ISSAC 2020: 45th International Symposium on Symbolic and Algebraic Computation*, Kalamata, Greece, July 2020.

[73] Niels Moller and Torbjörn Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60:165–175, 2011.

[74] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[75] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[76] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[77] Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In Georges Gonthier and Michael Norrish, editors, *3rd International Conference on Certified Programs and Proofs (CPP)*, volume 8307 of *Lecture Notes in Computer Science*, pages 66–81, Melbourne, Australia, December 2013.

[78] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer Science & Business Media, 2002.

[79] Mário José Parreira Pereira. *Tools and Techniques for the Verification of Modular Stateful Code.* PhD thesis, Université Paris-Saclay, December 2018.

[80] Alexandre Peyrard, Nikolai Kosmatov, Simon Duquennoy, and Shahid Raza. Towards formal verification of contiki: Analysis of the AES–CCM* modules with Frama-C. In *RED-IOT 2018 - Workshop on Recent advances in secure management of data and resources in the IoT*, 2018.

[81] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. In *22nd ACM SIGPLAN International Conference on Functional Programming*, September 2017.

[82] David L. Rager, Jo Ebergen, Dmitry Nadezhin, Austin Lee, Cuong Kim Chau, and Ben Selfridge. Formal verification of division and square root implementations, an Oracle report. In *16th Conference on Formal Methods in Computer-Aided Design*, pages 149–152, October 2016.

[83] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[84] Raphaël Rieu-Helft. A Why3 proof of GMP algorithms. *Journal of Formalized Reasoning*, 12(1):53–97, 2019.

[85] Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond. How to get an efficient yet verified arbitrary-precision integer library. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, volume 10712 of *Lecture Notes in Computer Science*, pages 84–101, Heidelberg, Germany, July 2017.

[86] David M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, 1999.

[87] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and symbolic computation*, 6(3-4):289–360, 1993.

[88] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 398–414, 2005.

[89] Marc Schoolderman. Verifying branch-free assembly code in Why3. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 66–83, 2017.

[90] Natarajan Shankar. A brief introduction to the PVS2C code generator. In *AFM@ NFM*, pages 109–116, 2017.

[91] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.

[92] Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963.

[93] Alan Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating-machines*, 1949.

[94] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

[95] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in Coq. *Journal of Functional Programming*, 25, 2015.

[96] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. Cryptology ePrint Archive, Report 2017/536, 2017. `https://eprint.iacr.org/2017/536`.

ÉCOLE DOCTORALE
Sciences et technologies
de l'information
et de la communication (STIC)

Titre : Développement et vérification de bibliothèques d'arithmétique entière en précision arbitraire

Mots clés : Analyse statique, Vérification déductive, Why3, Arithmétique entière en précision arbitraire, Preuve par réflexion

Résumé : Les algorithmes d'arithmétique entière en précision arbitraire sont utilisés dans des contextes où leur correction et leurs performances sont critiques, comme les logiciels de cryptographie ou de calcul formel. GMP est une bibliothèque d'arithmétique entière en précision arbitraire très utilisée. Elle propose des algorithmes de pointe, suffisamment complexes pour qu'il soit à la fois justifié et difficile de les vérifier formellement. Cette thèse traite de la vérification formelle de la correction fonctionelle d'une partie significative de GMP à l'aide de la plateforme de vérification déductive Why3. Afin de rendre cette preuve possible, j'ai fait plusieurs ajouts à Why3 qui permettent la vérification de programmes C. Why3 propose un langage fonctionnel de programmation et de spécification appelé WhyML. J'ai développé des modèles de la gestion de la mémoire et des types du langage C. Ceci m'a permis de réimplanter des algorithmes de GMP en WhyML et de les vérifier formellement. J'ai aussi étendu le mécanisme d'extraction de Why3. Les programmes WhyML peuvent maintenant être compilés vers du C idiomatique, alors que le seul langage cible était OCaml auparavant. La compilation de mes programmes WhyML résulte en une bibliothèque C vérifiée appelée WhyMP. Elle implémente de nombreux algorithmes de pointe tirés de GMP en préservant presque toutes les astuces d'implémentation. WhyMP est compatible avec GMP, et est comparable à la version de GMP sans assembleur écrit à la main en termes de performances. Elle va bien au-delà des bibliothèques d'arithmétique en précision arbitraire vérifiées existantes. C'est sans doute le développement Why3 le plus ambitieux à ce jour en termes de longueur et d'effort de preuve. Afin d'augmenter le degré d'automatisation de mes preuves, j'ai ajouté à Why3 un mécanisme de preuves par réflexion. Il permet aux utilisateurs de Why3 d'écrire des procédures de décision dédiées, formellement vérifiées et qui utilisent pleinement les fonctionnalités impératives de WhyML. À l'aide de ce mécanisme, j'ai pu remplacer des centaines d'annotations manuelles de ma preuve de GMP par des preuves automatiques.

Title: Development and verification of arbitrary-precision integer arithmetic libraries

Keywords: Static analysis, Deductive verification, Why3, Arbitrary-precision integer arithmetic, Proof by reflection

Abstract: Arbitrary-precision integer arithmetic algorithms are used in contexts where both their performance and their correctness are critical, such as cryptographic software or computer algebra systems. GMP is a very widely-used arbitrary-precision integer arithmetic library. It features state-of-the-art algorithms that are intricate enough that their formal verification is both justified and difficult. This thesis tackles the formal verification of the functional correctness of a large fragment of GMP using the Why3 deductive verification platform. In order to make this verification possible, I have made several additions to Why3 that enable the verification of C programs. Why3 features a functional programming and specification language called WhyML. I have developed models of the memory management and datatypes of the C language, allowing me to reimplement GMP's algorithms in WhyML and formally verify them. I have also extended Why3's extraction mechanism so that WhyML programs can be compiled to idiomatic C code, where only OCaml used to be supported. The compilation of my WhyML algorithms results in a verified C library called WhyMP. It implements many state-of-the-art algorithms from GMP, with almost all of the optimization tricks preserved. WhyMP is compatible with GMP and performance-competitive with the assembly-free version. It goes far beyond existing verified arbitrary-precision arithmetic libraries, and is arguably the most ambitious existing Why3 development in terms of size and proof effort. In an attempt to increase the degree of automation of my proofs, I have also added to Why3 a framework for proofs by reflection. It enables Why3 users to easily write dedicated decision procedures that are formally verified programs and make full use of WhyML's imperative features. Using this new framework, I was able to replace hundreds of handwritten proof annotations in my GMP verification by automated proofs.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France