Chapter 1

Definitions

1.1 Language

1.2 Termination

We work with terms written in the System FR formulation. In System FR, correctness is shown via a typing algorithm. So if we manage to type a term we automatically get its partial correctness (i.e. its verification conditions) and its total correctness (i.e. termination).

Definition 0.1 (Termination)
Termination of higher-order function means that when this function is fully applied then the resulting term reduces to a value in a finite number of steps.

Note that higher-order arguments are also assumed to be terminating for this definition.

Definition 0.2 (Problem)
A problem given by a function f is a set of functions S built from the reflexive transitive closure of f transformed into the strong connected components of its call-graph. One rules out not well-formed datatypes, non-recursive components and unckecked functions.

Chapter 2

Current state

2.1 The semantics of the decrease constructor

I'm taking the semantics of decreases to be that for a given function f, when I call recursively in f to f, the measure decreases. This ignores calls to other functions.

2.2 Processing pipeline

2.3 Processors

2.3.1 Recursion processor

The recursion processor was taken as is from previous work Voirol [2013]. The algorithm checks the body of a SCC with a single function:

$$f(\overrightarrow{p_0}) = \dots g^{(i)}(\overrightarrow{p_i}) \dots f^{(j)}(\overrightarrow{p_j}) \dots$$

Each $g^{(i)}$ is checked for termination. If they terminate, the algorithm checks $f^{(j)}$. If $j = 0$ then $m(f) = 0$. Otherwise, we search for a k where for all j we have $(\overrightarrow{p_j})_k < (\overrightarrow{p_0})_k$. In the current version, this decrease is just an increase in the field selectors. If this succeeds then $m(f) = (\overrightarrow{p_0})_k$.

2.3.2 Relation processor

The relation processor computes for each function f its function calls f_i, adding relations $f \xrightarrow{p_i} f_i$. It then computes all $\text{cmp}(size(\overrightarrow{p_j}), size(\overrightarrow{p}))$. The assumption is that $\forall j.\text{cmp}(size(\overrightarrow{p_j}), size(\overrightarrow{p})) \in \{\leq, <\}$.

$$m(f) = \begin{cases} (size(\overrightarrow{p}), 0) & \forall j.\text{cmp}(size(\overrightarrow{p_j}), size(\overrightarrow{p})) =< \\ (size(\overrightarrow{p}), i) & \forall j. f \xrightarrow{p_j} f_j \implies m(f_j).2 < i \end{cases}$$

2.3.3 Application strengthener

Functions are ordered in reverse topological order, i.e. first we have the functions that don't call other functions, then those that call these and so on. For each function, we scan its body:

$$f(\overrightarrow{p_0}) = \text{if } (path_i)\ f^{(i)}(\overrightarrow{p_i}) \ldots \text{if } (path_j)\ f^{(j)}(\overrightarrow{p_j}) \ldots$$

and compute constraints $f \xrightarrow{c} v$ where v is one of f's higher-order parameters and $c \in \{<, \leq, =\}$. The constraint is derived from three pieces of information:

1. For any application $v(\overrightarrow{p})$ we compute $\text{cmp}(size(\overrightarrow{p}), size(\overrightarrow{p_0}))$. We write $v \to$.

2. Recursively, for any function invocation $f_i(\ldots, v, \ldots)$ in the body of f, we know constraints $f_i \to v$ signaling the decrease in parameters in applications of v inside f_i. Note that functions invocations in the same strong connected component as f won't have any constraint.

3. We take into account the decrease in parameters from f to f_i, i.e. $\text{cmp}(size(\overrightarrow{p_i}), size(\overrightarrow{p_0}))$. We write $f \to f_i$.

Finally, we can compute $f \to v$ as:

$$f \to v = \begin{cases} ? & v \to =? \\ ? & f_i \to v =?, f_i \text{ unanalyzed} \\ ? & f \to f_i =?, f_i \text{ unanalyzed} \\ ? & f_i \to v =<, f \to f_i =? \\ < & f_i \to v =<, f \to f_i \neq? \\ f \to f_i & f_i \to v =\leq \end{cases}$$

2.3.4 Postcondition strengthener

As in the application strengthener, functions are ordered in reverse topological order so that we can profit from previously found post-conditions. Consider a function:

$$f(\overrightarrow{p}) = \{v \Rightarrow pre(v)\}\{b\}\{v \Rightarrow post(v)\}$$

the algorithm checks (in the reverse topological order) if the following stronger post-condition holds:

$$f(\overrightarrow{p}) = \{v \Rightarrow pre(v)\}\{b\}\{v \Rightarrow post(v) \wedge \mathrm{cmp}(size(v), size(\overrightarrow{p}))\}$$

in which case it is stored for further iteration of the algorithm, i.e. when we repeat the process on a bigger function, we first encode the solved functions with the strengthened post-conditions and then we call the solver to test if the new post-condition holds. Note that cmp is just the order relation of the current processor.

2.3.5 Chain processor

There are only two types of chains:

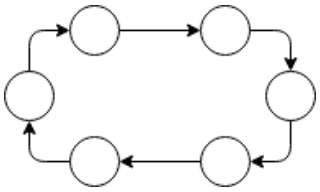

Figure 2.1: No loops chain

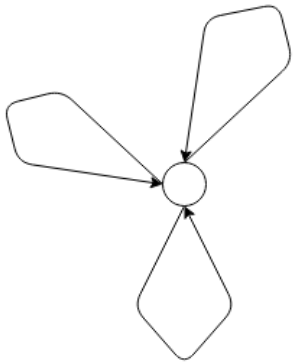

Figure 2.2: Loop chain

The processor then behaves as follows. Take a chain:

$$c = f_0 \overset{p_1}{\to} f_1 \overset{p_2}{\to} \ldots \overset{p_n}{\to} f_n \overset{p_0}{\to} f_0$$

the underlying solver checks:

$$\forall \overrightarrow{p} \in \text{Param}(f_0).p_1 \wedge \ldots \wedge p_n \wedge p_0 \implies \text{m}(f_n \cdot \ldots \cdot f_0(\overrightarrow{p})) < \text{m}(\overrightarrow{p})$$

i.e. for each logic path that leads to the above chain of calls and each parameter tuple $\overrightarrow{p}$ triggering such logic path, we have $m(f_n(\ldots f_1(f_0(\overrightarrow{p})))) < m(\overrightarrow{p})$.

Definition 0.3 (Loop point)
loops$(f_0) = \{f.f \neq f_0 \wedge f_0 \to_* f \to_+ f \to_* f_0 \wedge$ *there is no occurrence of* f_0 *in the segment between*

Definition 0.4 (Chain)
chains$(f_0) = \{c.f_0 \to_* f_0\}$

Theorem 1
The chain processor is sound.

Proof:
Let's proof that if chain processor outputs *terminates* on f_0 then f_0 is terminating. There are two cases:

- loops $= \emptyset$: then we start analysing on f_0 and there is a single chain. Consider f_i on that chain. We want to prove that termination of the

chain starting from f_0 is equivalent to termination of the chain starting from f_i.

If computation starts from f_i. For non-termination, I have to give an input where the path condition starting from f_0 is not true. If from this input we stay in the chain and reach f_0 then we will reenter the chain and decrease the magnitude. So we terminate. Thus, we have to leave the chain at some point f_j.

- If we leave the chain to call a function g out of the SCC then the termination burden is on g.
- If we leave the chain to call recursively f_j, then f_j would be a loop point. This is a contradiction.
- If we leave the chain to call some f_t with $0 < t < j$, we would have that f_j is again a loop point. That's a contradiction. Note that loop points are unaware of path conditions.
- If we leave the chain to call some f_t with $j < t <= n$ then I repeat my argument on f_t.

Since the length of the chain is finite, I either eventually reach f_0 or call a function g that is external to the computation. So g carries the burden of the termination proof.

- loops $\neq \emptyset$: then we start analysing on the unique loop point l and show decrease in each chain. Here the analysis is complicated by the fact that we can unfold chains to prove a decrease.

Does the chain termination behaviour depend on the start of the chain? There are two cases:

In particular, this paths only cover the set of potentially non-terminating parameter values. We see thus, that there is a fundamental difference between building a decreasing measure for a chain and proving that the chain terminates: the termination argument needs to be shown decreasing even for parameter tuples that don't trigger fully the chain. [1]

Our goal is to give measure functions $m_0, \ldots, m_n$ such that:

$$\forall i \in \{0..n\}.\forall \overrightarrow{p} \in \mathrm{Param}(f_i).p_{i+1} \implies \mathrm{m}_{(i+1) \bmod n}(f_i(\overrightarrow{p})) < \mathrm{m}_i(\overrightarrow{p})$$

Then set:

$$m_0(\overrightarrow{p}) = (m(\overrightarrow{p}), 0, n)$$

$$m_1(\overrightarrow{q}) = \begin{cases} (\mathrm{m}(f_n \cdot \ldots \cdot f_1(\overrightarrow{q})), 1, n-1) & \text{if } p_1 \wedge p_2 \wedge \ldots \wedge p_0 \\ (m(\overrightarrow{p}), 0, n-1) & \text{otherwise} \end{cases}$$

$$m_2(\overrightarrow{q}) = \begin{cases} (\mathrm{m}(f_n \cdot \ldots \cdot f_2(\overrightarrow{q})), 1, n-1) & \text{if } (\exists \overrightarrow{j} \in \mathrm{Param}(f_0).p_1 \wedge \overrightarrow{q} = f_0(\overrightarrow{j})) \wedge p_2 \wedge \ldots \wedge p_0 \\ (m(\overrightarrow{p}), 0, n-2) & \text{otherwise} \end{cases}$$

$$\ldots$$

Assume function f_i was called with parameter tuple $\overrightarrow{q}$. $\overrightarrow{q}$ can either trigger the full of c or only a subchain of it. The path condition to complete the chain p_i and the final arguments arg_i calling f_0 can be computed easily from the representation of c. Thus if p_i holds, I annotate the measure $(m(\mathrm{arg}_i), (n-i) \bmod n, n-i)$.

Conclusion to investigate:

Both examples of chain processor failing require an unfolding of chains. However, it is not clear that we took care of that during the design of the measures.

2.3.6 Loop processor

[1] Moreover, this reinforces the advantage of having the semantics of decrease to account only for loops that start and end in the same function. In that case the decrease only refers to logic conditions that are clear from the original function.

Chapter 3

The transformers hierarchy

Chapter 4

Problematic examples

Here we will list some termination problems that are unsolved.

4.1 Relation processor

4.1.1 Indirect example

```
def app(f: BigInt => BigInt, arg: BigInt): BigInt = f(arg)
def f(x: BigInt): BigInt =
 if (x > 0) app(f,x-1) else BigInt(0)
```

Figure 4.1: Indirect.scala

Semantics perspective

According to chapter 1, *app* is terminating since its parameter f is assumed to be so when fully applied. f is terminating since when fully applied, the two if branches are terminating.

Processor perspective

The termination pipeline sees that *app* is non-recursive and so considers it terminating. The termination of f is shown by the relation processor using the sum of argument sizes. There are two relations:

$$(fd : f40, p : x114 > 0, fi : app0(f40, x114 - 1), il : false)$$

$$(fd : f40, p : x114 > 0 \wedge |x115| <= |x114 - 1|, fi : f40(x115), il : true)$$

the second relation corresponds to the call to $f(x)$ which appears after η-expanding f. The extra path condition is inferred in the application strengthener.

Type checker perspective

The type checker sees that the inferred measure on f is non-decreasing and thus declares it invalid. In the body of f we have a call to f:

$$app(y \Rightarrow fy, x - 1)$$

If m is the inferred measure, the type checker emits the verification condition $m(x) > m(y)$ for arbitrary x and y. Therefore there is no m that can show termination of the example to the current typechecker.

Proposed solution

We have an example of a semantically terminating function without measure.

If we want to keep the semantics of type checker, we could specify the extra information that the application strengthener gives as a refinement type as shown below. The type of *app* would then need to be generalized (maybe in a polymorphic manner) via a program transformation.

```
def app(f: BigInt => BigInt, arg: BigInt): BigInt = f(arg)

def f(x: BigInt): BigInt = {
 if (x > 0) app((y: { z : BigInt | z <= x-1 }) =>
    f(y),x-1) else BigInt(0)
}
```

I believe that this would probably handle 5 ignored tests: ConstantPropagation, HOTermination, Indirect, QuickSort, MergeSort and MergeSort2.

4.1.2 Modules hanging

ls is Nil || ls.t is Nil ||

ListPrimitiveSize[BigInt](filter[BigInt](ls.t, (x1 : $BigInt$) => $x1$ < ls.h)) < ListPrimitiveSize[BigInt](ls)

ls is Nil || ls.t is Nil || size[BigInt](filter[BigInt](ls.t, (x1 : $BigInt$) => $x1$ < ls.h)) < size[BigInt](ls)

Chapter 5

Modifications

5.1 Recursion processor

We have added a variant that checks that there is always a decrease in some parameter in the integer ordering. This already solves Ex2:

```
fastExpA(base: BigInt, exp: BigInt): BigInt =
 require(exp >= 0)
 if (exp == 0)  1
 else fastExpA(base+1, exp-1)
```

This can affect negatively performance, since we need to call the underlying solver. Yet, the underlying equations should be easy to automate. We have observed that a general size change termination would solve the Nested14 benchmark:

```
rec1(j) = rec2(j,j)
rec2(k,j) = if (k > 0) rec1(j-1) else 0
```

5.2 Strengthener

A first observation is that the RelationBuilder violates the unique responsability principle. Why should it build relations and inject some extra information on them.

So we try to decouple both functionalities. We try to annotate the trees with the learnt metadata. We can annotate the parameters of a call. However, we cannot annotate in the termination checker, the function that is

called, since this might be unknown to the strongly connected component and one queries the termination checker with a single function. So this last meta-data information can only be annotated by the interface. In a method called refineSignature. The information is passed when a component is cleared.

Besides, the high-level reflection is that there is no good-handling of higher-order termination with the size-change termination analysis.

Chapter 6

Reflection

6.0.1 La esencia de la computación

¿Cuál es la esencia de la computación? En la actividad que se desarrolla en un facultad de informática, parece que lo fundamental es el *dato* y la transformación que ocurre sobre este dato. De aquí surgen dos preguntas naturales:

- ¿Qué datos son admisibles?
- ¿Qué transformaciones son admisibles?

La respuesta a esta pregunta ha llevado al desarrollo de modelos: el lambda-cálculo, las máquinas de Turing y la teoría de la recursión, que han sido demostrados equivalentes. De aquí, la tesis de Church y Turing que establecería que cualquier modelo que representa lo que los humanos podemos calcular mediante procedimientos mecánicos tendría un poder equivalente al de estas teorías.

Así que la práctica en la computación consiste en reducir aquellos problemas a algunos de estos lenguajes. Así, la lógica puede ser codificada en términos de funciones recursivas, una metodología empleada por Göedel en la prueba del teorema de incompletitud. Y en la teoría de la (meta)programación se reducen los objetos a elementos del lambda cálculo. De modo que, podríamos convenir que las implementaciones que existen en nuestros días son fieles a esta forma de trabajo y no pueden superarla.

Nuestro tema de estudio es precisamente es precisamente una de esas cuestiones que muestran las limitaciones de estos modelos. No puede de-

scribirse una transformación en este modelo, que dado un conjunto de transformaciones secuenciales escrito en forma finita permita determinar si este conjunto de transformaciones puede ser interpretado en tiempo finito. No está demás dar una intuición de por qué esto es así:

Sea f una tal transformación y sea

$$g(i) = f((g, i));\ \text{if(yes) don't terminate else terminate}$$

está transformación hará justamente lo contrario de lo que prediga f. Lógicamente, esta explicación require especificar los datos y las transformaciones admisibles según el sentido anterior. Ahora bien, la indecibilidad del problema de la parada solo informa la incapacidad de un sólo algoritmo para exhauscir el espacio de los programas terminantes. Surgen entonces algunas preguntas:

- ¿podría existir una clase de algoritmos que exhausciera el espacio de los programas terminantes? Ciertamente, esta clase no podría ser finita pues entonces podríamos probar cada algoritmo en paralelo. ¿Existe algún cardinal de algoritmos que lo permitiera?

- ¿Hay algún problema cuya terminación es indecidible?

De estas reflexiones llegamos al punto inicial de nuestra investigación: la clasificación de los problemas terminantes y no terminantes. Aquí podemos utilizar cualquier técnica de la matemática moderna para llevar a cabo dicha clasificación. Para ello hay que encontrar *codificaciones* adecuadas así como valernos del conocimiento experto en cada área.

Veamos entonces, qué elementos habría que pensar aquí. Respecto a las codificaciones:

- ¿Cambia algo el hecho de que el número de transformaciones sea 1, finito o un número ilimitado de transformaciones?

- en el espíritu de la lógica es natural reducir problemas mecánicos a problemas numéricos. Este es el enfoque de Göedel pero también de Turing en su tesis de doctorado donde muestra como los problemas numéricos y los computacionales guardan una estrecha relación. Este es el enfoque por ejemplo del size-change termination.

- Existen otras codificaciones como los sistemas de reescritura de términos.

Respecto a las herramientas para la clasificación:

- teorías matemáticas que separan objetos como la topología que separa en abiertos y cuya base son los conjuntos por el simple hecho de que la matemática de aquel tiempo se basaba en la teoría de conjuntos.
- problemas de clasificación son habituales en machine learning. Otras técnicas probabilistas podrían ser de uso, haciendo sampling del algoritmo con distintos parámetros podría verse su trayectoria en un cierto ordinal (piénsese la trayectoria en $\mathbb{N} \times \mathbb{N}$).

Respecto a las aplicaciones. Nos planteamos si un intérprete podría no terminar cuando se ejecuta a sí mismo. También nos preguntamos qué diferencia existe entre lo que se puede calcular y lo que es cierto, esto es, la lógica constructiva.

6.0.2 La clave no es el resultado sino cómo se calcula el resultado

En la teoría de la recursividad lo importante no es la funciones que se pueden calcular, sino cómo se pueden calcular. Esto es ya visto en la base, ciertos tipos de funciones básicas se pueden calcular mientras que otras funciones se obtienen combinando estas mediante ciertos combinadores. Esto es precisamente lo que limita el poder de lo que se puede expresar en un computador. Esto es la esencia de la ciencia de los compiladores, lo importante no es lo que se escribe sino cómo se ejecuta lo que se escribe.

En nuestro trabajo utilizaremos dos formalismos: el lambda-cálculo y teoría de la recursividad. No está claro aún cuál es el papel de las medidas sobre la terminación de programas.

- ¿es posible quedarse con una sola función para analizar la terminación? sí.

  Si f, g son funciones mutuamente recursivas podemos poner $h(w) = if(w = 1)thencdigodefelsecdigodeg$ donde se ha hecho $unfold(f)[f \mapsto h(1), g \mapsto h(0)]$.
- podemos expresar el procesador de relaciones y el procesador de cadenas como transformaciones de programas
- ¿podríamos reducir el número de parámetros de las funciones de Gö

Bibliography

N. Voirol. Termination Analysis in a Higher-Order Functional Context. Master's thesis, School of Computer and Communication Sciences, EPFL, 2013.