1 Introduction

In the last lecture we introduced arithmetic refinements to capture some internal invariants of data structures, including, for example, characterizing their size. This information is critical to capture computational complexity of functions or, in our case, processes. The key goal is to capture this information in types, using the kind of conceptual tools and techniques we have developed so far.

We carry out our development in the context of message-passing concurrency, capturing the total amount of work that a configuration of communicating processes does. This may also be called the sequential complexity of the program, because it is the complexity if all operations are performed one at a time. Its counterpart is the span, or the parallel complexity, which arises from the assumption that all operations that can be done in parallel (on an abstract machine with arbitrarily many processors), will in fact be done in parallel. Once both work and span are known one can derive some bounds on running time based on the number of available processors using Brent’s Theorem [Bre74]. Because the last true lecture in this course is cancelled in favor of a seminar, we encourage you to read up on how to think about parallel complexity in this context [DHP18a]. The development we present in this lecture is a reformulation of a prior ergometric type system [DHP18b], close to the one implemented in Rast [DP20].

The ingredients that come together to give a sound ergometric type system are the following:
L26.2 Ergometric Types

Cost Model: We do not want an abstract type system to depend on the particulars of a machine or a compiler. Instead we want to parametrize the ergometric types with a cost model. Depending on the model chosen by a programmer or language designer, concrete cost may be different but the type system will remain the same.

Potential: When analyzing code we do not “count up” to track the cost, but we provide each function or process with some potential \( q \geq 0 \) and have it “count down” the remaining potential. This simplifies the statics and the dynamics in the formalization.

Transfer of Potential: A process may decide not to perform some operations itself, but instead transfer some of its potential to other processes. This allows us to perform amortized analysis, which is a common and powerful technique for characterizing the computational complexity of a function or process.

Linearity: Once potential is invested in processes or data structures, it is important that it not be duplicated. This is where the nature of substructural type systems in general and linear ones in particular comes to the rescue because it prevents just such duplication.

For all these reasons we chose message-passing concurrency as presented in Lecture 24 as our basis, but it could easily be adapted to linear functional or linear shared memory code (Lecture 23). For nonlinear programs we need what we called a quasi-linear type system in Mini-Project 2.1 or sharing in Resource-Aware ML (RAML, see [HAH12]).

2 Cost Model

The cost model is represented by a transformation from an original process \( P \) to a new process \( P' \) that insert instances of the construct \( \text{work } w \). As an example, consider the cost model where each send operation costs 1 erg (= 1 unit of work). Then we transform every process \( x.V \) to \( \text{work } 1 ; x.V \). As another example, in a model where each call (recursive or not) costs 1 erg, we would transform each call \( x \leftarrow f y_1 \ldots y_n \) to \( \text{work } 1 ; x \leftarrow f y_1 \ldots y_n \).

We omit from the cost model the index terms and quantifiers introduced for expressions in the last lecture. That’s because we think of their purpose as static, more precisely capturing properties of the data and functions we work with, rather than dynamic, contributing to the outcome of computation itself.
For this lecture we remain within the cost model where every send costs 1 erg. Because every channel is linear, every send is matched by a receive (except perhaps at the top level), so also counting receives would just double the cost.

In addition to the predefined costs, the programmer may also insert work \( n \) explicitly. This turns out to be useful in two circumstances. One is that different branches of a process require different amount work, in which case the programmer may equalize them. Another is that the programmer may wish to choose the “free” cost model where all operations are free, and then insert explicit work \( w \) to mark whatever they would like to count.

3 Statics

Isolating work in a separate construct work \( w \) has the nice consequence that we update the rule in a simple and systematic way to track the potential available to a process. The new judgment has the form

\[
\Delta \vdash^q P :: (x : \tau)
\]

which means that process \( P \) with potential \( q \) uses the channels in \( \Delta \) (according to their type) and provides channel \( x : \tau \). The rules are derived from those in the Lecture 23 Rule Set, except that variables are no longer annotated as read or write since we are working in a message-passing setting here.

The one new rule is the one that consumes potential by doing work.

\[
\frac{\Delta \vdash^q P :: (x : \tau)}{\Delta \vdash^{w+q} \text{work } w ; P :: (x : \tau)} \quad \text{tp/work}
\]

Because \( w \) and \( q \) must be natural numbers, for this rule to apply there must be at least \( w \) potential available. Otherwise, the process \( \text{work } w ; P \) cannot be typed.

The remaining rules are derived in a systematic way from the prior rules. For rules with zero premises, we require the potential to be 0. This is because we want to track the potential exactly, instead of an upper bound. For example:

\[
\frac{\cdot \vdash^0 x.() :: (x : 1)}{\text{send/unit}} \quad \frac{y : \tau \vdash^0 x \leftarrow y :: (x : \tau)}{\text{forward}}
\]
The rule that spawns a new process must split the potential between the two processes.

\[
\Delta \vdash^r P :: (x : \tau) \quad \Delta', x : \tau \vdash^q Q :: (z : \sigma)
\]

\[
\Delta, \Delta' \vdash^{r+q} (x \leftarrow P ; Q) :: (z : \sigma)
\]

spawn

In the notation we note the potential that is imparted on the freshly spawned process. This is not strictly necessary, but it simplifies the dynamics.

For the remaining rules, the potential is preserved to all premises. For internal choice (⊕) or external choice (⊗) this may be surprising at first, but remember that at runtime exactly one branch will be chosen, based on the message received. This branch should have the full potential of the case: no more and no less. Here are two example rules:

\[
\Delta, y_i : \tau_i \vdash^q P_i :: (z : \sigma) \quad \text{(for all } i \in I) \quad \text{recv/label}
\]

\[
\Delta, x : \oplus_{i \in I}(i : \tau_i) \vdash^q \text{case } x (i \cdot y_i \Rightarrow P_i)_{i \in I} :: (z : \sigma) \quad \text{recv/channel}
\]

4 Dynamics

The dynamics is adapted from the one in Lecture 24 on message-passing concurrency. Process objects carry potential, while messages do not. Since all the potentials are checked statically, before the program is ever executed, these potential annotations are strictly necessary, but they play a key role in the proofs of progress and preservation.

Configurations \( \mathcal{C} ::= \text{proc}^q P \mid \text{msg} c V \mid \mathcal{C}_1, \mathcal{C}_2 \mid (\cdot) \)

The rules adapt straightforwardly, with the first rule being new.

\[
\text{proc}^w+q \ (\text{work } w ; P) \quad \Rightarrow \quad \text{proc}^q P
\]

\[
\text{proc}^{r+q} \ (x \leftarrow P ; Q) \quad \Rightarrow \quad \text{proc}^q ([c/x]P), \text{proc}^q ([c/x]Q) \quad \text{(spawn; c fresh)}
\]

\[
\text{proc}^0 \ (c.V) \quad \Rightarrow \quad \text{msg} c V \quad \text{(send)}
\]

\[
\text{msg} c V, \text{proc}^q \ (\text{case } c K) \quad \Rightarrow \quad \text{proc}^q (V \triangleright K) \quad \text{(receive)}
\]

\[
\text{msg} d V, \text{proc}^0 \ (c \leftarrow d) \quad \Rightarrow \quad \text{msg} c V \quad \text{(pos. forward)}
\]

\[
\text{proc}^0 \ (c \leftarrow d), \text{msg} c V \quad \Rightarrow \quad \text{msg} d V \quad \text{(neg. forward)}
\]
Note that dynamically a process would not be able to make a transition if it didn’t have sufficient potential to do the work. Therefore, the progress theorem expresses that there will always be sufficient potential. The preservation theorem expresses that the overall potential in a configuration plus the amount of work performed remains invariant. The transitions cannot create new potential out of thin air, but they also cannot drop any existing potential. See Exercise 1 for further thoughts.

5 Example: Lists

As a first example we consider the cost of list constructors. We ignore here the cost of fold, primarily because the prior work [DHP18a, DP20] uses so-called equirecursive types where a type is considered equal with its unfolding so no fold message is actually necessary. It is not difficult to update the examples with such messages.

\[ \text{list } \alpha = (\text{nil} : 1) \oplus (\text{cons} : \alpha \otimes \text{list } \alpha) \]

A prior, we would expect the empty list to require 2 messages (nil and \langle \rangle), which is also the case for cons (cons and \(x\)).

First the regular types.

\[ \alpha \text{ type } \vdash \text{nil} :: (l : \text{list } \alpha) \]
\[ l \leftarrow \text{nil } \alpha = l' \leftarrow l'.\langle \rangle ; \]
\[ l.(\text{nil} \cdot l') \]

\[ \alpha \text{ type, } x : \alpha, l : \text{list } \alpha \vdash \text{cons} :: (k : \text{list } \alpha) \]
\[ k \leftarrow \text{cons } \alpha x l = k' \leftarrow k'.\langle x, l \rangle ; \]
\[ k.(\text{cons} \cdot k') \]

Next we apply the cost model. We also annotate the process definitions with the potential these processes require.

\[ \alpha \text{ type } \vdash^2 \text{nil} :: (l : \text{list } \alpha) \]
\[ l \leftarrow^2 \text{nil } \alpha = l' \leftarrow^1 \text{work } 1 ; l'.\langle \rangle ; \]
\[ \text{work } 1 ; l.(\text{nil} \cdot l') \]

\[ \alpha \text{ type, } x : \alpha, l : \text{list } \alpha \vdash^2 \text{cons} :: (k : \text{list } \alpha) \]
\[ k \leftarrow^2 \text{cons } \alpha x l = k' \leftarrow^1 \text{work } 1 ; k'.\langle x, l \rangle ; \]
\[ \text{work } 1 ; k.(\text{cons} \cdot k') \]

It is easy to verify that the these definitions and types for the definitions are ergometrically correct.
6 Exploiting Arithmetic Refinements

In the next example we will assign a type to the \textit{append} process for two lists. Clearly, this should be proportional to the size of the first list \( l_1 \), because when we reach its end we just forward to the second list \( l_2 \).

\[
\text{list } α \, n = (\text{nil} : n = 0 ∧ 1) ⊕ (\text{cons} : n > 0 ∧ α ⊗ \text{list } α(n - 1))
\]

We propose the following type

\[
α \, \text{type}, n : \mathbb{N}, m : \mathbb{N}, l_1 : \text{list } α \, n, l_2 : \text{list } α \, m \parallel 2^n \quad \text{append} :: (k : \text{list } α(n + m))
\]

because the \textit{append} process will have to send two messages (\textit{cons} and an element \( x \)) for each element of the list \( l_1 \) which has length \( n \). Indeed, this is easy to check. We have annotated lines with the known arithmetic constraints (in blue) and also the remaining potential (in red).

\[
k \overset{2^n}{\leftarrow} \text{append } α \, n \, m \, l_1 \, l_2 = \quad \% 2n
\]

\[
\begin{align*}
\text{case } l_1 (\text{nil} \cdot ⟨⟩) & \Rightarrow k \leftarrow l_2 & \% n = 0, 2n = 0 \\
| \text{cons} \cdot ⟨x, l'_1⟩ & \Rightarrow & \% n > 0, 2n
\end{align*}
\]

\[
\begin{align*}
k' \overset{2^((n - 1) - 1)}{\leftarrow} \text{append } α \, (n - 1) \, m \, l'_1 \, l_2 ; & \% 2n - 2(n - 1) = 2 \\
k \overset{2}{\leftarrow} \text{cons } α \, ((n - 1) + m) \, x \, k' & \% 2 - 2 = 0
\end{align*}
\]

As a second, similar example we consider \textit{reverse} which calls \textit{revapp} as an auxiliary process. \textit{revapp} moves all the elements from the first list onto the second (the accumulator), to be returned at the end.

\[
α \, \text{type}, n : \mathbb{N}, m : \mathbb{N}, l : \text{list } α \, n, acc : \text{list } α \, m \parallel 2^n \quad \text{revapp} :: (k : \text{list } α(n + m))
\]

\[
k \overset{2^n}{\leftarrow} \text{revapp } α \, n \, m \, l \, acc = \quad \% 2n
\]

\[
\begin{align*}
\text{case } l (\text{nil} \cdot ⟨⟩) & \Rightarrow k \leftarrow acc & \% n = 0, 2n = 0 \\
| \text{cons} \cdot ⟨x, l'⟩ & \Rightarrow & \% n > 0, 2n
\end{align*}
\]

\[
\begin{align*}
acc' \overset{2}{\leftarrow} \text{cons } α \, m \, x \, acc ; & \% 2n - 2 \\
k \overset{2^{(n - 1)}}{\leftarrow} \text{append } α \, (n - 1) \, (m + 1) \, l' \, acc' & \% 2n - 2 - 2(n - 1) = 0
\end{align*}
\]

\[
α \, \text{type}, n : \mathbb{N}, l : \text{list } α \, n \parallel 2^n + 2 \quad \text{reverse} :: (k : \text{list } α \, n)
\]

\[
k \overset{2^{n + 2}}{\leftarrow} \text{reverse } α \, n \, l = \quad \% 2n + 2
\]

\[
\begin{align*}
acc \overset{2}{\leftarrow} \text{nil } α ; & \% 2n \\
k \overset{2^n}{\leftarrow} \text{revapp } α \, (n + 0) \, l \, acc & \% 0
\end{align*}
\]


7 Transferring Potential

A key operation in amortized analysis is to be able to transfer potential between functions or processes. For this purpose we need two new type operators: one to send and one to receive potential (taking the point of view of the provider). We write \( \sq{q} \tau \) for sending potential \( q \) (a positive type) and \( \ll{q} \tau \) for receiving potential \( q \) (a negative type).

\[
\begin{align*}
\text{Types} & \quad \tau \quad ::= \quad \ldots | \: \sq{q} \tau | \: \ll{q} \tau \\
\text{Small Values} & \quad V \quad ::= \quad \ldots | \: \text{pot } q \; x \\
\text{Continuations} & \quad K \quad ::= \quad \ldots | \: (\text{pot } q \; x \Rightarrow P)
\end{align*}
\]

The potential \( q \) in the receiving continuation is not a variable: we must be able to predict statically, via the type, how much potential is being received. Due to the duality between positive and negative types, we only have one new small value and one new continuation. However, we need new rules since the generic ones do not account for potential transfer.

\[
\begin{align*}
\text{proc}^q (c. (\text{pot } q \; c')) & \quad \mapsto \quad \text{msg} \; c \; (\text{pot } q \; c') \\
\text{msg} \; c \; (\text{pot } q \; c'), \text{proc}^r (\text{case} \; c \; (\text{pot } q \; x \Rightarrow P)) & \quad \mapsto \quad \text{proc}^{r+q} (\lfloor c'/x \rfloor P)
\end{align*}
\]

The generic rules from Section 4 remain the same.

8 Example: A Binary Counter

Incrementing a binary counter may flip as few as one bit (the lowest, if it is 0) and as many as there are bits in the number (if they are all 1). Amortized analysis tells us that \( n \) increments starting from 0 will flip at most \( 2n \) bits. In this section we have the type-checker prove this for us.

We represent each bit in the binary number by a process, \( \text{bit0} \) or \( \text{bit1} \), plus one process \( \text{emp} \) for the end of the bit string. The client interacts with the lowest bit, sending it \( \text{inc} \) messages (eliding other parts of the interface that may be present). Every message by a process corresponds to exactly one bit flip, so counting the number of messages as our default cost model does is appropriate. Ignoring potential transfer, the type would be a an external choice with just one alternative.

\[
\text{ctr} = (\text{inc} : \text{ctr}) \& ()
\]

The key idea of this amortized analysis is that each \( \text{bit1} \) process maintains one unit of potential that it can use to send the carry bit when it is incremented. Furthermore, the client sends not only the increment message, but
an additional unit of potential. If the lowest bit is a bit0, it is flipped to bit1 and the extra unit stored. If the lowest bit is bit1 it becomes a bit0, sends the increment message representing the carry, plus the required additional unit of potential. In code:

\[
\begin{align*}
ctr &= (\text{inc} : a^1 ctr) \\
y : ctr \parallel^0 \text{bit0} :: (x : ctr) \\
y : ctr \parallel^1 \text{bit1} :: (x : ctr) \\
&\cdot \parallel^0 \text{emp} :: (x : ctr)
\end{align*}
\]

\[
x \leftarrow \text{bit0} y = \\
\cases{x \ (\text{inc} \cdot x' \Rightarrow) \\
\cases{x' \ (\text{pot} 1 x'' \Rightarrow) \\
\quad x'' \leftarrow \text{bit1} y})
\]

\[
x \leftarrow \text{bit1} y = \\
\cases{x \ (\text{inc} \cdot x' \Rightarrow) \\
\cases{x' \ (\text{pot} 1 x'' \Rightarrow) \\
\quad y' \leftarrow \text{work} 1 ; y.(\text{inc} \cdot y')} \\
\quad y'' \leftarrow y'.(\text{pot} 1 y'') \\
\quad x'' \leftarrow \text{bit0} y''}
\]

\[
x \leftarrow \text{emp} = \\
\cases{x \ (\text{inc} \cdot x' \Rightarrow) \\
\cases{x' \ (\text{pot} 1 x'' \Rightarrow) \\
\quad y \leftarrow \text{emp} \\
\quad x'' \leftarrow \text{bit1} y})}
\]

Type-checking these definitions prove our theorem. Every increment message costs the clients two units of potential (1 to send the message, and 1 to be stored as internal potential). Since every increment message corresponds one bit being flipped, and each message send costs 1 erg, overall there are less than \(2^n\) bits being flipped. More precisely, if the \(k\) bits 1 in the representation, then \(2^n - k\) bit will have been flipped (because each of the bits 1 will store one unit of potential that has not been used yet to flip a bit). Except for zero, there will always be at least one bit 1 in the counter, so the total number of bit flips is in fact bounded by \(2^n - 1\) for \(n > 0\).
Exercises

Exercise 1 Formulate the properties of progress and preservation for the ergonomic type system, as mentioned at the end of Section 4. Sketch the key steps in these proofs as they pertain to potential and work.

Exercise 2 Revisit the example of queues, as in Exercise L22.3 and Section L24.7.

(i) Give an ergonomic definition of stacks with push and pop operations.

(ii) Give an ergonomic definition of stack reversal.

(iii) Give an ergonomic definition of queues with enqueue and dequeue operations, implemented as a bucket brigade.

(iv) Give an ergonomic definitions of queues using two stacks. If this necessitates additional ergonomic types, or new types for processes such as stack reversal, please state these revised definitions explicitly.

References


