坑点一览

前端

不要用工厂模式！！！

- 使用纯粹的工厂模式+语法树的前端架构会导致错误处理时不能因地制宜地向父节点传递信息，造成错误处理难以进行

后端

不要企图实现$\mathcal{O}(n)$的数据流分析

- 不考虑 `break` 和 `continue` 和 `goto` 时数据流图是仙人掌图，因此可以把循环回边标记为轻边，然后在重边生成的拓扑序上跑两遍迭代完成$\mathcal{O}(n)$数据流分析。但加入 `break`，`continue` 后这种做法的正确性被破坏，导致必须重构。

关于SSA中 `phi` 节点的处理

- 显然phi节点最终需要在前一个基本块中翻译出相应的处理代码，完成寄存器的重新分配。考虑处理时的局面，存在三种移动：

  1. 寄存器到栈
  2. 寄存器到寄存器
  3. 栈到寄存器

  必须按照123的顺序处理，否则就是错的。另外，考虑寄存器到寄存器这一阶段，显然数学上可以看成是一个置换，对这个置换建图会搞出来一个基环外向树森林。由于环的存在，不能简单地按照拓扑序的逆序做寄存器移动，而要先找到环，然后插入一个临时寄存器再移动。

  这部分代码极容易出锅，并且由于实际编译时绝大部分时间都是很简单的情形，加上这部分即使真的出错也很难查错，强烈建议进行单独且全面的测试。

  ~~（本人就是过掉了所有的辅助评测和竞速点，然后在优化末期忽然发现这里有bug）~~

优化

迷一样的复写优化

- 如果按照以下方法生成SSA，那么是原生带有复写优化的：

  > 1. 将源程序中的变量看作助记符，把四元式的结果看作真变量。
  > 2. 维护原变量当前绑定到哪个四元式上，并据此生成每个基本块的SSA
  > 3. 做数据流分析，消除不必要的SSA

  然而，在神秘的第三步中，很容易忘记删除因为来源相同而冗余的phi节点，进而导致复写优化的部分失效。在分支很多的程序中，这种失效十分明显，有可能指令数会多个0。目标代码上具体表现为给phi节点生成了大量无效的处理指令。

函数调用的性能魔咒

- 如果按照以下方式处理函数调用，那么需要特别小心：

  > 1. 给所有的函数加上4个没用的参数
  > 2. 这样就可以无视前4个参数必须放在 `$a0 ~ $a3` 的限制
  > 3. 于是所有参数都放在栈上传递

  这在每次函数调用时都会多 `2*LS*count` 条指令，LS是单条访存指令的实际开销，count是参数个数。这在函数体短，且出现循环或递归调用时影响显著，指令数可能会多1～2倍。

  一个好的处理是做伪内联，就是把函数调用看作特殊的跳转，把参数看成phi节点，然后像处理普通跳转一样处理它。

小锅

无效跳转：

- 显然，如果一个 `j` 指令刚好跳到下一条指令，那么可以删除；显然，如果一个 `b` 或 `j` 指令刚好跳到下一条 `j` 指令，那么可以优化。这非常的显然，however，如果不小心干了这件事，你会发现你不小心进入了玄学领域：

  > 1. 不小心把跳到 `jal` 的跳转语句优化掉了
  > 2. 然后发现不仅没有出错甚至跑得飞快

个人认为是测试点不够强导致的。能够把 `jal` gank掉的情况基本只能是函数调用是if或while块里的第一条语句并且调用该函数不需要保护现场。这种情况最可能出现在递归调用的最后一层，而此时将 `jal` gank掉不仅确实不出错而且可以提前返回上一级函数，一举多得。只可惜它是错的。

常量传播：

- 写常量传播的时候一开始很可能会忘记写补偿代码，然后导致一些循环条件之类的该成立没成立，然后因此莫名其妙地删掉了一些无用循环使得代码跑得飞快。只可惜它也是错的。