دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

پایان‌نامه کارشناسی ارشد

گرایش مهندسی نرم‌افزار

عنوان

رویکردی مبتنی بر زبان برای پیش‌گیری از رقابت داده

نگارش

علی قنبری

استاد راهنما

مهران سلیمان فلاح

مهر ۱۳۹۳

به نام خدا

تعهدنامه اصالت اثر

تـا ریـخ:

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

اینجانب علی قنبری متعهد می‌شوم که مطالب مندرج در این‌پایان نامه حاصل کار پژوهشی اینجانب تحت نظارت و راهنمایی اساتید دانشگاه صنعتی امیرکبیر بوده و به دستاوردهای دیگران که در این پژوهش از آن‌ها استفاده شده است مطابق مقررات و روال متعارف ارجاع و در فهرست منابع و مآخذ ذکر گردیده است. این پایان نامه قبلاً برای احراز هیچ مدرک هم‌سطح یا بالاتر ارائه نگردیده است.

در صورت اثبات تخلف در هر زمان، مدرک تحصیلی صادر شده توسط دانشگاه از درجه اعتبار ساقط بوده و دانشگاه حق پیگیری قانونی خواهد داشت.

کلیه نتایج و حقوق حاصل از این پایان‌نامه متعلق به دانشگاه صنعتی امیرکبیر می‌باشد. هرگونه استفاده از نتایج علمی و عملی، واگذاری اطلاعات به دیگران یا چاپ و تکثیر، نسخه‌برداری، ترجمه و اقتباس از این پایان‌نامه بدون موافقت کتبی دانشگاه صنعتی امیرکبیر ممنوع است. نقل مطالب با ذکر مآخذ بلامانع است.

علی قنبری

تقدیم به پدر، مادر، و نیایش عزیزم...

تقدیر و تشکر

لازم است از معلم اخلاق و استاد بزرگوارم جناب آقای دکتر مهران سلیمان فلاح که در این دوره دو ساله راهنمایی علمی مرا بر عهده گرفته بودند و پیش از این، در دوره کارشناسی، با ارائه درس زبان‌های برنامه‌سازی مرا به تحقیق در حوزه علوم کامپیوتر نظری علاقه‌مند کردند نهایت تقدیر و تشکر را داشته باشم.

بر خود واجب می‌دانم که از پدر و مادر بزرگوارم، به خاطر زحمت‌های بی‌دریغشان، تشکر کنم. پدرم در این دوران هزینه تحصیل مرا فراهم کرده و در سختی‌ها دعاگوی بنده بود. مادرم همواره برای سلامت و موفقیتم دعا می‌کند و اگر در روزهای سخت دوره کارشناسی ارشد نصیحت‌های مادرانه ایشان نبود من دلسرد می‌شدم و نمی‌توانستم این پایان‌نامه را به پایان برسانم.

از تمام افرادی که در فراهم کردن برنامه‌های محک به من کمک کردند تشکر می‌کنم. در نهایت، از دوستان هم‌خوابگاهی خود که در این دوران برادرانه حامی و پشتیبان من بودند نهایت تقدیر و تشکر را دارم.

چکیده

گسترش استفاده از پردازنده‌های چند هسته‌ای در دستگاه‌های محاسباتی و نیاز بیش از پیش به استفاده بهینه از این منابع پردازشی باعث شده است که امروزه اکثر تولید کنندگان نرم‌افزار به برنامه‌سازی همروند روی آورند. اما برنامه‌سازی همروند نقص‌های جدیدی را متوجه سیستم‌های نرم‌افزاری می‌کند. رقابت داده از جمله این نقص‌ها است که در اثر همگام‌سازی نادرست ریسه‌ها در دسترسی به منابع مشترک به وجود می‌آید. با توجه به طبیعت غیرقطعی خطای رقابت داده، ردیابی محل دقیق و زمان وقوع آن کار دشواری است. از طرفی مشکلاتی که این خطا می‌تواند به وجود آورد در بسیاری از موارد فاجعه‌بار است. بنابراین، کشف و پیش‌گیری خودکار رقابت داده برای ارتقای کیفت نرم‌افزار از اهمیت ویژه‌ای برخوردار است.

با وجود پیشرفت‌های چشم‌گیر ابزارهای خودکار کشف رقابت داده در دهه اخیر، نیاز به یک ابزار مقیاس‌پذیر، و در عین حال درست و دقیق، هنوز برطرف نشده است. در این پایان‌نامه، روشی مبتنی بر زبان برای کشف رقابت داده ارائه و ایده‌های خود را در زیرمجموعه‌ای از زبان برنامه‌سازی جاوا و به عنوان نمونه‌ای از چارچوب یکنوا نمایش داده‌ایم. بر اساس این روش، می‌توان ابزارهای خودکار ایستای کشف رقابت داده مقیاس‌پذیر، درست، و دقیق پیاده‌سازی کرد. مقیاس‌پذیری در روش ما از طریق تحلیل‌های پیمانه‌ای فراهم شده است. رخدادهای پارامتری و مفهوم خلاصه متدها ایده اصلی ما برای ارائه تحلیل‌های پیمانه‌ای است. این روش، بر خلاف روش‌های موجود، معادلات جریان داده را برای کل برنامه تشکیل نمی‌دهد، بلکه هر متد را جداگانه تحلیل کرده و برای هر یک خلاصه‌ای پارامتری محاسبه می‌کند. در مکان‌های فراخوانی، به ازای هر متد فراخوانی شده، به جای محاسبه مجدد جواب برای متد مورد نظر، خلاصه مربوط به آن متناسب با متن فراخوانی نمونه‌سازی می‌شود. روش ارائه شده در این پایان‌نامه، قابلیت تحلیل برنامه‌های باز، مانند کتابخانه‌ها که برنامه استفاده کننده از آن‌ها در دسترس نیست، را دارد. در نهایت، با توجه به این که حاشیه‌نویسی صرفاً در واسط‌ها و نیز محل اعلان متدها، فیلدها، و متغیرها انجام می‌شود، مقدار حاشیه‌نویسی مورد نیاز با پیاده‌سازی‌های مختلف یک واسط تغییری نمی‌کند. به عنوان اثباتی برای صحت ادعای خود مبنی بر عملی بودن ساخت ابزاری مقیاس‌پذیر مبتنی بر روش ارائه شده، ابزاری برای وارسی برنامه‌های جاوا پیاده‌سازی کرده‌ایم. آزمایش‌های ما نشان می‌دهند که این پیاده‌سازی اولیه ضمن این‌که قابلیت تحلیل برنامه‌های بزرگ را دارد، دقت آن نیز قابل مقایسه با دقیق‌ترین ابزارهای موجود برای کشف رقابت داده است.

واژه‌های کلیدی:

رقابت داده، تحلیل جریان داده، رخداد، جاوا، گراف جریان کنترل، تحلیل اشاره‌گر، تحلیل جریان کنترل

فهرست عناوین

فهرست شکل‌ها

فهرست جدول‌ها

فهرست علائم

علائم لاتین

e	رخداد
R	نوع رخداد خواندن
W	نوع رخداد نوشتن
N	نوع رخداد ورود به مانیتور
X	نوع رخداد خروج از مانیتور
t	اسم نوع
fd	نام فیلد
mn	نام متد
c	نام کلاس

علائم یونانی

τ	شناسه ریسه
η	اجرای مجرد
π	تابع تصویر

فصل اول

مقدمه

رقابت داده نقصی است که در برنامه‌های همروند، در اثر همگام‌سازی نادرست ریسه‌ها، در دسترسی به منابع مشترک ایجاد می‌شود. همان‌طور که می‌دانیم یک برنامه را می‌توان به عنوان توصیفی از مجموعه‌ای از اجراها تعریف کرد. یک اجرا دنباله‌ای از رخدادها است، و هر رخداد عملی است که با اجرای یک دستورالعمل در برنامه اتفاق می‌افتد (برای مثال دسترسی به یک مکان حافظه یا اخذ یک قفل). دو رخداد دسترسی را متضاد گوییم هرگاه مربوط به یک مکان مشترک حافظه باشند، حداقل یکی از آن‌ها از نوع نوشتن باشد، و دسترسی‌ها توسط ریسه‌های مختلف صورت گیرد.

تعریف ۱.۱. هرگاه دو رخداد متضاد در یک اجرا موجود باشند که با هم رابطه پیش‌رویدادی ندارند گوییم یک خطای رقابت داده (در آن اجرا) اتفاق افتاده است. در صورتی که اجرایی از یک برنامه وجود داشته باشد که در آن حداقل یک خطای رقابت داده اتفاق افتد می‌گوییم آن برنامه دارای رقابت داده است. ■

لازم به یادآوری است که رابطه پیش‌رویدادی، در این تعریف، به عنوان یک ترتیب جزئی بر روی رخدادهای یک اجرا تعریف می‌شود. به زبان ساده رابطه پیش‌رویدادی هنگامی بین دو رخداد وجود دارد که این دو رخداد مربوط به یک ریسه باشند یا توسط یک مکانیزم همگام‌سازی ترتیبی بین آن‌ها اعمال شده است.

برای مثال، شبه‌کد زیر را در نظر بگیرید که در آن دو ریسه T1 و T2 قرار است به‌صورت همروند اجرا شوند.

```
//Thread T1 :                    //Thread T2 :
synchronized(l1){                synchronized(l2){
    e1.f = ...                       e2.f = ...
}                                }
```

شکل ۱.۱: بخشی از توصیف دو ریسه از یک برنامه

توجه کنید که در این مثال دستور synchronized(l){s} همان ساختار همگام‌سازی زبان برنامه‌سازی جاوا [۱] است. ریسه‌ای که این دستورالعمل را اجرا می‌کند، قبل از اجرای s، ابتدا قفلی بر روی شی نشان داده شده توسط عبارت l اخذ

کرده و بعد از اتمام اجرای s این قفل را آزاد می‌کند. یک اجرا از این برنامه به صورت زیر است که در آن تعدادی از رخدادهای مهم را فهرست کرده‌ایم.

رخدادهای مربوط به ریسه T1: $e_۱ = N(\texttt{T1}, o_{11}), e_۲ = W(\texttt{T1}, \langle o_{\texttt{e1}}, \texttt{f}\rangle), \ldots, e_۳ = X(\texttt{T1}, o_{11}),$

رخدادهای مربوط به ریسه T2: $e_۴ = N(\texttt{T2}, o_{12}), e_۵ = W(\texttt{T2}, \langle o_{\texttt{e2}}, \texttt{f}\rangle), \ldots, e_۶ = X(\texttt{T2}, o_{12}).$

در این مجموعه هر رخداد دارای یک شناسه است که با e_i، $۱ \leq i \leq ۶$، نشان می‌دهیم. همچنین، در این مجموعه سه نوع رخداد مشاهده می‌شود: رخدادهای N یا همان رخداد ورود به بلوک `synchronized`، رخدادهای W یا همان نوشتن بر روی یک فیلد یک شی (یک مکان حافظه)، و رخدادهای X یا همان رخداد خروج از بلوک `synchronized`. همان‌طور که مشاهده می‌شود هر رخداد حاوی اطلاعاتی مانند شناسه ریسه‌ی ایجاد کننده رخداد و نیز شی یا مکان حافظه‌ای که رخداد مورد نظر بر روی آن اتفاق افتاده است. نمادهای o_{11}، o_{12}، $o_{\texttt{e1}}$، و $o_{\texttt{e2}}$، به ترتیب، برای نشان دادن شی‌های محاسبه شده برای عبارت‌های l1، l2، e1، و e2 به‌کار می‌رود. دوتایی شی و نام فیلد نیز برای نشان دادن یک مکان حافظه مورد استفاده قرار گرفته است.

آن‌گونه که گفته شد، رخدادهای مربوط به یک ریسه با رابطه پیش‌رویدادی مرتب شده‌اند. همچنین، با توجه به اینکه یک ریسه صرفاً زمانی می‌تواند قفلی را اخذ کند که قفل مورد نظر در حالت آزاد باشد (یعنی در آن لحظه ریسه دیگری آن را اخذ نکرده باشد)، رخدادهای X از یک ریسه با رخدادهای N از ریسه‌های دیگر، زمانی که هر دو مربوط به یک شی باشند، رابطه پیش‌رویدادی دارند. با توجه به این مشاهدات رخدادهای $e_۱$ تا $e_۳$ (رخدادهای مربوط به ریسه T1) دو به دو باهم رابطه پیش‌رویدادی دارند. رخدادهای $e_۴$ تا $e_۶$ (رخدادهای مربوط به ریسه T2) نیز دو به دو باهم رابطه پیش‌رویدادی دارند. تنها در صورتی که o_{11} و o_{12} نشان دهنده یک شی واحد باشند می‌توان گفت که رخداد $e_۳$ پیش‌تر از $e_۴$ اتفاق افتاده است (یا $e_۶$ پیش‌تر از $e_۱$ اتفاق افتاده است، بسته به اینکه کدام ریسه زودتر قفل مشترک را اخذ کرده است). بنابراین، رخدادهای دو ریسه نسبت به هم مرتب می‌شوند و مستقل از اینکه آیا رخدادهای $e_۲$ و $e_۵$ متضاد هستند یا خیر می‌توان گفت که اجرا خالی از رقابت داده است. بر عکس زمانی که o_{11} و o_{12} اشیاء متمایزی هستند، رخدادهای دو ریسه با هم رابطه پیش‌رویدادی ندارند و یعنی در صورتی که رخدادهای $e_۲$ و $e_۵$ متضاد باشند خطای رقابت داده اتفاق می‌افتد.

همان‌طور که مشاهده کردیم رقابت داده از امکان اجرای غیر بسیط نواحی بحرانی در یک برنامه ناشی می‌شود. بنابراین، وجود این نقص ممکن است باعث بروز اشتباه در مقدار نهایی متغیرها (خروجی برنامه) شود. برای مثال در دو عملیات بانکی موازی برای واریز پول به یک حساب، ممکن است تاثیر یکی از عملیات نادیده گرفته شود. این‌گونه رفتارها در بسیاری از موارد تخطی از توصیف نیازمندی محسوب شده و در نتیجه نامطلوب است. رقابت داده در یک دستگاه پرتو درمانی به نام Therac-25 موجب مرگ پنج بیمار و زخمی شدن چندین نفر شد. خطای رقابت داده، همچنین، موجب از کار افتادن سامانه مدیریت انرژی و در نتیجه موجب خاموشی آمریکای شمالی در سال ۲۰۰۳ شد [۲]. از جمله سایر مشکلاتی که رقابت داده در یک برنامه به وجود می‌آورد می‌توان به امکان درست نبودن (عدم حفظ معنا) بهینه‌سازی و در حالت کلی ترجمه خودکار برنامه‌های دارای رقابت داده اشاره کرد [۳].

بنابراین، مقابله با رقابت داده از اهمیت ویژه‌ای برخوردار است ولی ارائه روشی بهینه کار دشواری است. به همین دلیل تلاش برای ارائه روش‌های مقابله با رقابت داده بیش از سی سال است که ذهن محققان را به خود مشغول کرده است. پژوهش ارائه شده در این پایان‌نامه نیز در این راستا قرار دارد.

با توجه به این‌که در این پایان‌نامه به بحث کشف رقابت داده پرداخته شده است در ادامه این فصل ابتدا مروری بر سابقه و وضعیت پژوهش در زمینه روش‌های کشف رقابت داده ارائه می‌شود. با مرور کمبودها در هر یک از پژوهش‌ها قادر خواهیم بود که صورت‌مسأله پژوهش خود را به‌طور صریح بیان کنیم. سپس در مورد راه حل خود، انگیزه ارائه آن، و میزان سهمی که در حل مشکلات بیان شده دارد بحث می‌کنیم. لازم به یادآوری است که از این به بعد منظور ما از رقابت، همان رقابت داده است.

۱.۱ روش‌های کشف رقابت‌داده

یک روش کشف رقابت آرمانی دارای دو ویژگی **درستی** و **تمامیت** است. درستی به این معنا است که در صورت وجود رقابت در یک برنامه، بررسی آن حداقل یک مورد وجود رقابت را گزارش دهد. تمامیت نیز به این معنا است که هر مورد گزارش مربوط به یک رقابت داده‌ی واقعی و امکان‌پذیر در برنامه باشد. به عبارتی دیگر، خاصیت درستی مربوط به میزان محافظه‌کار بودن در تشخیص وجود رقابت، و تمامیت مربوط به میزان دقت در تشخیص درست و اجتناب از ارائه گزارشات نادرست است. متأسفانه تحلیل‌های درست و تام (دارای خاصیت تمامیت) در حالت کلی محاسبه‌پذیر نیستند [۴]. بنابراین، در مواردی که هر یک از این خواص کمتر مورد توجه هستند نادیده گرفته می‌شوند. برای مثال، ابزارهای کشف رقابت درستی وجود دارد که تام نیستند (تعداد زیادی از گزارشات ارائه شده توسط این ابزارها نادرست است)، و بر عکس ابزارهای کشف رقابت تامی موجود است که درست نیستند (ممکن است برخی از موارد رقابت داده‌ی واقعی و امکان‌پذیر نادیده گرفته شود.)

در حالت کلی دو نوع روش برای کشف رقابت داده وجود دارد: روش‌های **پویا** (یا مبتنی بر دنباله آثار)، و روش‌های **ایستا**. در ادامه این بخش سابقه و وضعیت این دو نوع از روش‌ها را بررسی می‌کنیم.

ورودی یک الگوریتم کشف رقابت داده که با استفاده از روش‌های مبتنی بر دنباله آثار پیاده‌سازی شده است، همان‌طور که از نام آن پیدا است، دنباله‌ای از رخدادها بوده که در طول یک اجرای یک برنامه اتفاق افتاده‌اند. به این روش‌ها پویا نیز می‌گویند چرا که با بخش پویای (در زمان اجرای) برنامه‌ها در ارتباط هستند. روش‌های پویا را در حالت کلی می‌توان با استفاده از دو مفهوم **رابطه پیش‌رویدادی** و یا **مجموعه قفل‌ها** پیاده‌سازی کرد. شاخه‌ای از پژوهش که بر روی روش‌های پویا انجام می‌شود پیشرفت زیادی کرده است، به‌گونه‌ای که در سال‌های اخیر محققان با معرفی تعاریفی پیش‌رفته‌تر از رابطه پیش‌رویدادی قادر به ارائه الگوریتمی شده‌اند که تنها با یک بار اجرای برنامه بر روی یک ورودی خاص تمامی موارد رقابت داده به ازای آن ورودی را در عرض چند ثانیه گزارش می‌دهد [۵، ۴]. قبل از مرور کارهای انجام شده در زمینه کشف رقابت داده لازم است این دو مفهوم را بیشتر معرفی کنیم. لازم به یادآوری است که در این پایان‌نامه جز اندکی در مورد روش‌های پویا بحث نمی‌کنیم.

رابطه **پیش‌رویدادی** لَمپورت [۶]، یک ترتیب جزئی غیربازتابی بر روی رخدادهای یک سیستم توزیعی است. برای شناسایی رقابت داده، کافی است جفت رخدادهای متضادی که **فاقد** رابطه پیش‌رویدادی هستند شناسایی شوند. ابتدا نیاز داریم تعریفی دقیق‌تر از رابطه پیش‌رویدادی، متناسب با مقیاس کارهای انجام شده مورد علاقه ما، ارائه دهیم.

در صورتی که $\langle e_0, e_1, \dots \rangle$، دنباله‌ای (احتمالاً نامتناهی) از رخدادها (یک اجرا) مفروض باشد، رابطه پیش‌رویدادی کوچکترین رابطه‌ای است که در این شرایط صدق می‌کند: (۱) اگر دو رخداد e_i و e_j مربوط به یک ریسه بوده و $i < j$ باشد، آنگاه $e_i \to e_j$، (۲) اگر دو رخداد e_i و e_j متعلق به ریسه‌های متمایز باشند، به‌طوری که e_i رخداد آزادسازی یک قفل و e_j رخداد اخذ همان قفل بوده و $i < j$ باشد، آنگاه $e_i \to e_j$، و (۳) رابطه $\to$ دارای خاصیت تعدی است.

لازم به یادآوری است که در تعریف رابطه پیش‌رویدادی فرض کرده‌ایم که هر نمونه از اجرای یک دستورالعمل با یک رخداد متفاوت در دنباله رخدادها شناسایی می‌شود (به عبارتی دیگر تمامی رخدادهای موجود در یک اجرا دو به دو با هم متفاوت هستند). توجه کنید که در این تعریف اخذ و آزادسازی یک قفل را به ترتیب با دریافت و ارسال یک پیام در تعریف اصلی جای‌گزین کرده‌ایم.

حال باتوجه به رابطه پیش‌رویدادی، می‌توان رخدادهایی که در به وجود آمدن یک رقابت داده مشارکت دارند را شناسایی کرد: دو رخداد متمایز e_i و e_j از یک اجرا موجب بروز (خطای) رقابت داده می‌شود هرگاه: (۱) رخدادها مربوط به دسترسی به یک مکان مشترک حافظه باشند، (۲) حداقل یکی از آن‌ها مربوط به نوشتن در آن مکان باشد، و (۳) رخدادها با هم رابطه پیش‌رویدادی نداشته باشند (یعنی $e_i \not\rightarrow e_j$ و $e_j \not\rightarrow e_i$) [۴، ۷، ۶].

یک الگوریتم پویای کشف رقابت مبتنی بر رابطه پیش‌رویدادی، برای صحت‌سنجی یک برنامه، در حالت کلی این کارها را انجام می‌دهد: (۱) ساخت رابطه پیش‌رویدادی برای رخدادهایی که در برنامه تولید می‌شوند، (۲) یک تاریخچه دسترسی برای هر یک از منابع مشترک ساخته می‌شود. آمار رخدادهای دستکاری کننده یک منبع در تاریخچه دسترسی‌های مربوط به آن منبع ضبط می‌شوند، (۳) به ازای هر منبع مشترک، و به ازای هر جفت رخداد موجود در تاریخچه دسترسی مربوط به آن منبع، شرط بالا برای عدم وقوع رقابت داده بررسی می‌شود [۴]. توجه کنید که با توجه به پیچیدگی محاسباتی و فضایی بالای محاسبه گام (۱) به جای محاسبه دقیق رابطه، آن را تخمین می‌زنند یا به دلیل نادرست بودن رابطه پیش‌رویدادی، در حالت کلی، در روش‌های پویا از یک رابطه پیش‌رفته‌تر استفاده می‌کنند.

دشواری در تخمین دقیق رابطه پیش‌رویدادی، و نادرست بودن آن، موجب به وجود آمدن شاخه دیگری از پژوهش شده است که در آن به جای استفاده از رابطه زمانی بین رخدادهای برنامه‌ها از مفهوم **سیاست قفل‌گذاری** استفاده می‌شود. سیاست قفل‌گذاری به این صورت است که هر منبع مشترک همواره با استفاده از یک قفل یکتا محافظت می‌شود، و هر ریسه قبل از دسترسی به یک منبع باید قفل منتسب به آن منبع را اخذ کند. واضح است که در صورت پایبند بودن به این سیاست هیچ‌گونه رقابتی به وجود نمی‌آید. صحت‌سنجی برای اطمینان از پیروی برنامه‌ها از سیاست قفل‌گذاری، اساس روش‌های مبتنی بر مجموعه قفل‌ها را تشکیل می‌دهد. رویکرد کشف رقابت پویا مبتنی بر مجموعه قفل‌ها، اولین بار در [۸] استفاده شد. ابزارهای کشف رقابت که با این رویکرد پیاده‌سازی می‌شوند، می‌توانند با یک‌بار اجرای برنامه به ازای یک ورودی خاص تمامی موارد رقابت داده به ازای آن ورودی را گزارش دهند، اما از آنجایی که ترتیب زمانی بین رخدادهای متضاد در نظر گرفته نمی‌شود ممکن است دقت این دسته از روش‌ها بسیار پایین باشد. برای مثال، مقداردهی اولیه منابع مشترک یا دستکاری منابع مشترکی که در بازه‌ای از طول عمر خود به‌صورت محلی و توسط یک ریسه بخصوص مورد دسترسی قرار می‌گیرند، معمولاً بدون اخذ قفل انجام می‌شود. در مقاله [۸] ایده‌های اولیه بهبود دقت روش‌های مبتنی بر مجموعه قفل‌ها ارائه شده است، این ایده‌ها در مقاله [۹] با معرفی الگوهای بیشتر دسترسی به منابع مشترک گسترش داده شده‌اند. در نهایت، [۱۰] با ارائه الگوریتمی که به‌طور همزمان از مزایای دو روش مبتنی بر مجموعه قفل‌ها و رابطه پیش‌رویدادی استفاده می‌کند، نشان می‌دهد که می‌توان کمبودهای هر یک از روش‌ها را با ترکیب آن‌ها جبران کرد.

در روش‌های ایستا، الگوریتم با بررسی متن برنامه‌ها یا فایل باینری آن‌ها اقدام به کشف رقابت می‌کند. بنابراین، این دسته از روش‌ها ما را قادر می‌سازند که برنامه‌ها را قبل از این‌که راهی محل استفاده شوند و بدون اجرای آن‌ها وارسی کنیم، که این برای بسیاری از کاربردها مفید است [۱۱]. روش‌های ایستا را در حالت کلی می‌توان به سه دسته روش‌های **کاربردی**، روش‌های **مبتنی بر تحلیل جریان داده**، و روش‌های **مبتنی بر نوع** طبقه‌بندی کرد [۴، ۱۲]. البته دسته‌ای از روش‌ها نیز با فراهم آوردن ساختارهای زبانی سطح بالا مانند مانیتورها [۱۳] از وقوع رقابت جلوگیری می‌کنند. معمولاً به این گونه

راه‌کارهای ایستا که با گسترش زبان هدف همراه است، روش‌های سطح زبانی یا **مبتنی بر زبان** گفته می‌شود [۱۴].

روش‌های ایستای کاربردی از روش‌های اکتشافی برای کشف رقابت استفاده می‌کنند. در این روش‌ها، متن برنامه‌ها برای یافتن موارد تخطی از الگوهای معمول (بدون رقابت) برنامه‌سازی مورد جستجو قرار می‌گیرد. روش‌های کاربردی را تجربی یا ابتکاری نیز می‌گویند، چرا که از دیدگاه کشف و پیش‌گیری از رقابت داده نه درست هستند و نه تام. اما تجربه نشان داده است که این روش‌ها در عمل بسیار مفید واقع شده‌اند، و علاوه بر آن در استفاده از این روش‌ها در بسیاری از موارد نیازی به بررسی دقیق متن برنامه‌ها نیست، که این باعث می‌شود روش‌های کاربردی بسیار سریع باشند. برای مثال، یک الگو برنامه‌سازی همروند در زبان برنامه‌سازی جاوا این است که دسترسی به هر متغیر عمومی در داخل یک بلوک `synchronized` انجام می‌شود. تخطی از چنین الگوی برنامه‌سازی به یک الگوی اشکال معروف است. در یک الگوریتم اکتشافی برای زبان جاوا چنین الگوی اشکالی جستجو می‌شود، و برای مثال هنگامی که بدنه متدی حاوی یک دسترسی به یک متغیر عمومی بدون بلوک همگام‌سازی باشد یک مورد رقابت داده گزارش می‌شود [۴]. موفق‌ترین نمونه از کاربرد این دسته از روش‌ها ابزار RacerX [۱۴] است. این ابزار از یک تحلیل حساس به جریان و حساس به زمینه برای کشف رقابت و بن‌بست در برنامه‌های بزرگ سیستمی استفاده می‌کند. البته لازم به یادآوری است که حساس به جریان و زمینه بودن تحلیل مشکلی در مقیاس‌پذیری ابزار به وجود نمی‌آورد، زیرا چیزی که این تحلیل‌ها محاسبه می‌کنند بسیار ساده بوده و مهم‌تر از آن هیچ تحلیل اشاره‌گر یا دگرنامی مورد استفاده قرار نگرفته است. همین سبک‌وزن بودن تحلیل‌ها باعث دقت پایین اطلاعات بدست آمده، و در نتیجه ارائه گزارشات نادرست بسیار زیادی، می‌شود. در این ابزار به جای استفاده از تعریف صوری رقابت داده از روش‌های ابتکاری و تجربی برای بررسی اینکه کدام قفل‌ها از کدام عملیات محافظت می‌کنند، کدام قطعه کدها ممکن است به‌صورت چندریسه‌ای اجرا شوند، یا کدام دسترسی‌ها به منابع مشترک می‌تواند خطرناک باشد استفاده می‌شود. همچنین، به کمک روش‌های وزن‌دهی و آماری گزارشاتی که جدی بودن آن‌ها محتمل‌تر است به عنوان گزارش نهایی اعلام می‌شوند. ابزار RacerX بسیار سریع بوده و قادر است در مدت زمانی کمتر از ۱۵ دقیقه برنامه‌های چند میلیون خطی را تحلیل کند. آزمایش‌ها نشان می‌دهد که نتایج بدست آمده از استفاده از این ابزار بسیار امیدوار کننده است. سرعت و دقت بالای ابزار آن را مناسب برای توسعه به منظور استفاده در صنعت کرده است [۱۱]. ابزارهای Lint و Warlock از جمله ابزارهای قدیمی و شناخته شده کشف رقابت برای زبان C هستند که اهدافی هم‌سو با اهداف RacerX دارند [۱۵، ۱۱].

در سال‌های اخیر، سیستم FindBugs [۱۶] برای کشف رقابت داده و سایر نقص‌های نرم‌افزاری برای زبان جاوا ارائه شد. این ابزار نیز همانند تمام ابزارهای کشف رقابت کاربردی موجود متن برنامه‌ها را برای یافتن موارد تخطی از الگوهای بدون نقص و توصیه شده جستجو می‌کند. بنابراین، درست نیست. با این حال استفاده از این ابزار رضایت کاربران زیادی در صنعت را جلب کرده است.

روش‌های ایستای مبتنی بر جریان داده از تحلیل‌های جریان داده [۱۷] برای کشف رقابت استفاده می‌کند. با استفاده از تحلیل‌های جریان داده می‌توان الگوریتم‌های کشف رقابت درست و در بالاترین حد ممکن دقت (برای یک ابزار ایستا) ارائه داد [۴]. اما بدیهی است که دقت بالا در کشف رقابت تأثیرات منفی بر روی مقیاس‌پذیری یا سرعت خواهد داشت [۴، ۱۲]. الگوریتم ارائه شده در [۱۸] از یک تحلیل جریان داده‌ی حساس به جریان و حساس به زمینه برای کشف رقابت در برنامه‌های شی‌گرا استفاده می‌کند. این الگوریتم در اصل یک پیاده‌سازی ایستا از روش کشف رقابت مجموعه قفل‌ها است، با این تفاوت که تاثیر شروع (و نه پایان) ریسه‌ها نیز در نظر گرفته شده است. به عبارتی بهتر الگوریتم یاد شده به‌گونه‌ای ترتیب زمانی رخدادهای تولید شده توسط هر یک از ریسه‌ها را در نظر می‌گیرد. اما ساخت و بررسی گراف‌هایی که در تشریح این الگوریتم معرفی شده‌اند، از لحاظ زمان و فضا، پیچیده بوده و مقیاس‌پذیر نیست. این الگوریتم بعداً اساس کار یکی از سریع‌ترین و

دقیق‌ترین ابزارهای کشف رقابت برای زبان جاوا، یعنی Chord [۱۹، ۲]، قرار گرفت. تفاوت اساسی الگوریتم مورد استفاده در Chord با الگوریتم ارائه شده در [۱۸] این است که Chord از یک تحلیل اشاره‌گری حساس به زمینه استفاده می‌کند. اساس کار این ابزار به این صورت است که جفت دستورالعمل‌های مشکوک به شرکت در رقابت در یک برنامه با گذر از مراحلی پالایش می‌شوند، به‌طوری که در هر مرحله دستورالعمل‌هایی که ایمن بودن اجرای همروند آن‌ها اثبات می‌شود از مجموعه مورد نظر حذف می‌گردند. دقت این ابزار بعداً با ارائه یک تحلیل دگرنامی خاص منظوره و دقیق‌تر تقویت شد [۲۰]. ابزار Chord درست بوده و دارای گزارشات نادرست کمی است، همچنین تلاش بسیاری برای افزایش مقیاس‌پذیری آن صورت گرفته است [۲]. با این حال این ابزار نیاز به وجود کل برنامه دارد و این می‌تواند در بسیاری از موارد محدود کننده باشد، و اصولاً مقیاس‌پذیر نیست [۴].

الگوریتم مورد استفاده در ابزار Relay [۲۱]، از یک تحلیل ایستای پیمانه‌ای (و در نتیجه مقیاس‌پذیر) مبتنی بر جریان داده استفاده می‌کند. این الگوریتم درست نیست و به نسبت اندازه کدهایی که تحلیل می‌کند تعداد گزارشات نادرست کمتری را ارائه می‌دهد. دقت این ابزار را می‌توان با در نظر گرفتن روابط زمانی بین رخدادهای برنامه‌ها تقویت کرد. ایده خلاصه متدهای مورد استفاده در این ابزار راهی مناسب برای افزایش مقیاس‌پذیری الگوریتم‌های تحلیل است.

تحلیل همروندی برای محاسبه رابطه may-happen-in-parallel یا همان MHP، بین دستورالعمل‌های برنامه‌ها، مورد استفاده قرار می‌گیرد. با استفاده از ترکیبی از یک تحلیل همروندی و یک تحلیل دگرنامی می‌توان رقابت‌های موجود در برنامه‌ها را کشف کرد [۴، ۷]. این ایده اولین بار در [۲۲] ارائه شد. تحلیل همروندی ارائه شده در این مقاله پیمانه‌ای بوده و امکان تحلیل مؤثر رویه‌ها و علی‌الخصوص رویه‌های بازگشتی را دارد. بعدها با پیاده‌سازی ابزاری غیردرست و غیرپیمانه‌ای در [۲۳]، این ایده‌ها برای زبان جاوا جامه عمل پوشانیده شدند. دقت و سرعت ابزار مورد نظر بالا است، اما مشکلاتی از قبیل عدم مقیاس‌پذیری و وابسته به زبان بودن تحلیل‌های ارائه شده همچنان پابرجا است.

یک راه برای کاهش سربار ناشی از تحلیل‌ها (و در نتیجه افزایش مقیاس‌پذیری) بالابردن ریزدانگی تحلیل است: در مورد رقابت داده، به جای جستجو برای رقابت بر روی مکان‌های حافظه (فیلدهای هر شی) می‌توان رقابت بر روی شی‌ها را مورد بررسی قرار داد [۴]. به‌طور معمول **رقابت شی** را به این صورت تعریف می‌کنند که یک برنامه دارای رقابت شی است، هرگاه متد یا متدهایی از یک شی توسط دو یا چند ریسه بدون اعمال هیچ‌گونه محدودیت همگام‌سازی فراخوانی می‌شوند. رقابت شی، شرط لازم برای وقوع رقابت داده است، اما وقوع هر رقابت شی‌ای الزاماً منجر به رقابت داده نمی‌شود. این ایده در ابزارهای ایستا [۲۴] و پویا [۹] برای کاهش سربار تحلیل و بررسی و کاهش گزارشات نادرست مورد استفاده قرار گرفته است.

آخرین دسته از روش‌های ایستا که بررسی می‌کنیم روش‌های مبتنی بر نوع [۲۵، ۲۶، ۲۷] است. مطلوب‌ترین ویژگی این دسته از روش‌ها پیمانه‌ای بودن (و در نتیجه مقیاس‌پذیری) آن‌ها است [۴، ۱۲]. ایده‌های اولیه کشف رقابت با استفاده نوع‌ها، با توسعه دو حساب همروند نوع‌ایمن، در [۲۸، ۲۹] ارائه شد. در این حساب‌ها قفلی به نوع هر یک از شی‌ها منتسب می‌شود، و سیستم نوع تضمین می‌کند که در هر مورد استفاده از شی‌ها قفل منتسب به نوع آن شی اخذ شده است. این ایده همان اعمال سیاست قفل‌گذاری است که پیش‌تر در ابزارهای پویا مورد استفاده قرار می‌گرفت. ایده حساب همروند ایمن بعدها به زبان برنامه‌سازی جاوا گسترش داده شد [۳۰]. این زبان جاوای گسترش یافته برنامه‌نویس را متحمل سربار حاشیه‌نویسی بالایی می‌کند، قدرت توصیف زبان حاصل شده نیز پایین است، همچنین استفاده از عملگرهای تبدیل نوع محدود شده است. سربار حاشیه‌نویسی را می‌توان با استفاده از یک ابزار جانبی که از یک الگوریتم بسیار ساده برای حدس حاشیه‌نویسی‌ها استفاده می‌کند برطرف کرد [۳۱]. البته بعداً ثابت شد که این الگوریتم ساده، و در حالت کلی حدس حاشیه‌نوشته‌ها در زبان‌هایی به شیوه‌های مشابه حاشیه‌نویسی می‌شوند، دارای پیچیدگی NP-Complete است [۳۲]. قدرت توصیف زبان ارائه شده در [۳۰]

جدول ۱.۱: خلاصه ویژگی‌های روش‌های موجود کشف رقابت

	دقت	درستی	مقیاس‌پذیری	قدرت توصیف	میزان حاشیه‌نویسی	برنامه‌های باز
پویای مبتنی بر رابطه پیش‌رویدادی	زیاد	خیر	محدود	زیاد	ندارد	خیر
پویای مبتنی بر مجموعه قفل‌ها	کم	خیر	زیاد	قفل	ندارد	خیر
پویای ترکیبی	زیاد	خیر	محدود	قفل	ندارد	خیر
ایستای کاربردی	زیاد	خیر	زیاد	زیاد	ندارد	خیر
ایستای مبتنی بر تحلیل جریان داده	زیاد	بلی	محدود	زیاد	ندارد	محدود
ایستای مبتنی بر نوع	کم	بلی	زیاد	قفل	زیاد	بلی

بعداً با ارائه ساختارهای زبانی عمومی‌تر تقویت شد [۳۳]، یک سال بعد از آن همان محققان با ارائه مفهموم نوع‌های مالکیت دقت زبان را هرچه بیشتر تقویت کردند، و به سیستم نوع ارائه داده شده قابلیت کشف بن‌بست‌ها را نیز اضافه کردند [۳۴]. این دو پژوهش اساس کار بسیاری از پژوهش‌های دیگر شد. لازم به یادآوری است که با وجود درست بودن تمامی زبان‌های معرفی شده سربار حاشیه‌نویسی و پیچیدگی محاسباتی بالای الگوریتم‌های استنتاج نوع همچنان مشکل اصلی این شاخه از پژوهش برای پیش‌گیری از رقابت داده است [۴، ۱۲].

ابزار Locksmith [۳۵]، یک سیستم کشف رقابت داده برای زبان برنامه‌سازی C است. این ابزار به منظور استخراج اطلاعاتی مانند اشتراک مکان‌های حافظه بین ریسه‌ها، وضعیت قفل‌ها در هر نقطه از برنامه، و جریان داده‌ها در برنامه از چندین سیستم نوع و اثر برای استخراج انواع قیدها استفاده می‌کند. تحلیل‌های مورد استفاده در ابزار Locksmith با توجه به این‌که حساس به زمینه هستند، دقیق است. همچنین، در ابزار یاد شده از روشی نوین و سریع برای حل دستگاه معادلات ساخته شده برای یک برنامه استفاده شده است. اما از آنجایی که Locksmith دستگاه معادلات را برای کل برنامه تشکیل می‌دهد، اساساً مقیاس‌پذیر نیست.

بحثی دقیق‌تر در مورد ابزارها و ایده‌های ارائه شده در این فصل و سایر کارهای مهم که هنوز در مورد آن‌ها بحث نشده است، در فصل مربوط به کارهای مرتبط قابل مشاهده است. جدول ۱.۱ مباحث ارائه شده در این بخش را به‌طور خلاصه بیان می‌کند. دقت کنید که هدف از ارائه این جدول بیان دقیق ویژگی‌های ذاتی هر دسته از روش‌ها نیست، بلکه حسی از میزان پیشرفت در نتیجه‌ی پژوهش‌های انجام شده تا کنون در هر دسته از روش‌ها را ارائه می‌دهد. برای مثال، هنگامی که در این جدول بیان می‌شود که میزان حاشیه‌نویسی برای روش‌های ایستای مبتنی بر نوع زیاد است، به این معنا است که پیاده‌سازی الگوریتم‌های استنتاج نوع برای سیستم‌های نوعی که تا کنون ارائه شده‌اند غیر عملی است. در نهایت لازم به یادآوری است که معیارهای یاد شده در جدول ۱.۱ معیارهایی شناخته شده برای مقایسه روش‌های مختلف کشف رقابت است [۱۲، ۱۹]. منظور ما از برنامه‌های باز در بین معیارهای فهرست شده برنامه‌هایی مانند توابع و کلاس‌های کتابخانه‌ای است که شیوه استفاده از آن‌ها به دلیل نبود برنامه مشتری (برنامه‌ای که از توابع و کلاس‌ها کتابخانه‌ای استفاده می‌کند) در دسترس نیست. همچنین، منظور ما از قدرت توصیف این است که روش مورد نظر توانایی بررسی چه اصطلاحات برنامه‌نویسی را دارد. در این‌جا قفل به این معناست که روش مورد نظر صرفاً توانایی صحت‌سنجی برنامه‌هایی را دارد که از روش قفل‌گذاری برای همگام‌سازی بین‌ریسه‌ای استفاده می‌کند، و زیاد به این معنا است که روش مورد نظر توانایی صحت‌سنجی اکثر اصطلاحات همگام‌سازی مرسوم را دارد.

۲.۱ صورت مسأله

همان‌طور که مشاهده کردیم، مقیاس‌پذیر نبودن، یعنی عدم توانایی وارسی برنامه‌های بزرگ، از جمله مشکلات روش‌های ایستای درست است. مشکل دیگری که هنوز پاسخ دقیقی برای آن داده نشده است، قابلیت تحلیل برنامه‌های باز، یعنی کلاس‌ها و توابع کتابخانه‌ای است. رفع این کمبودها در [۲، ۷] به عنوان کارهای آینده معرفی شده است، اما هنوز پاسخی برای آن‌ها وجود ندارد [۴]. در این پایان‌نامه، قصد داریم این دو مشکل را برای روش‌های ایستای مبتنی بر جریان داده حل کنیم.

۱.۲.۱ راه حل ارائه شده

با مرور کارهای انجام شده در زمینه کشف رقابت، مشاهده می‌کنیم که یک تحلیل همروندی به‌علاوه یک تحلیل دگرنامی (یا صورت کلی آن یعنی تحلیل شکل هیپ، و تحلیل اشاره‌گر) برای کشف رقابت، کفایت می‌کند [۷]. همچنین، مشاهده می‌کنیم که روش‌هایی که این رویکرد را پیش گرفته‌اند، دچار مشکلات محدودیت قدرت توصیف نشده‌اند، زیرا تحلیل‌های همروندی توانایی در نظر گرفتن هر دو اصطلاح همگام‌سازی با قفل‌گذاری، و نیز شروع و پایان ریسه‌ها را دارند. از سوی دیگر، وجود تحلیل‌ها شکل هیپ و فرار مقیاس‌پذیر [۳۶]، راه برای ارائه روش‌های کشف رقابت مقیاس‌پذیر و پیمانه‌ای را هموارتر می‌کند.

البته همان‌طور که در بخش قبل مشاهده کردیم، مقیاس‌پذیر بودن می‌تواند موجب سربار حاشیه‌نویسی بالا شود. در این پایان‌نامه، سعی کرده‌ایم تعادلی بین سربار حاشیه‌نویسی و مقیاس‌پذیر بودن ایجاد کنیم، بگونه‌ای که سربار بالای حاشیه‌نویسی برای دستیابی به مقیاس‌پذیری (مثلاً از طریق پیمانه‌ای بودن)، مانع از استفاده از الگوریتم‌های ارائه شده برای کدهای موروثی و کارهای صنعتی نشود (چیزی که مشکل اصلی روش‌های مبتنی بر نوع برای کشف رقابت است.) از آنجایی که تعدادی ساختار زبانی به زبان هدف یعنی زبان جاوا اضافه کرده‌ایم، این روش ایستا را یک روش مبتنی بر زبان نامیده‌ایم.

در نهایت، لازم به یادآوری است که این پژوهش اولین تلاش برای ساخت یک الگوریتم درست، با دقت بالا، مقیاس‌پذیر، و دارای حاشیه‌نویسی کم است. به‌طور خلاصه، نوآوری‌های این پایان‌نامه به شرح زیر است:

- یک تحلیل درست، با دقت بالا، مقیاس‌پذیر، با قابلیت تحلیل برنامه‌های باز، و دارای سربار حاشیه‌نویسی ثابت و معقول است. تحلیل مورد نظر نمونه‌ای از چارچوب یکنوا است. این تحلیل، برای برنامه‌های باز اطلاعات سودمندی را عرضه می‌کند. تحلیل ارائه شده، قابل پارامتری شدن است. به این معنا که می‌توان توجه الگوریتم را تنها بر روی فیلدهایی متمرکز کرد که نبود رقابت داده بر روی آن‌ها برای ما اهمیت دارد. این توانایی باعث کاهش تعداد گزارش‌های نادرست و امکان استفاده از ابزار پیاده‌سازی شده برای کدهایی را به وجود می‌آورد که در برخی از موارد وقوع رقابت‌های داده خوش‌خیم برای آن‌ها قابل تحمل است. موارد حاشیه‌نویسی در روش ارائه شده محدود به اعلان متدها و فیلدها است. بنابراین، تعداد حاشیه‌نوشته‌ها متناسب با تعداد اعلان فیلدها و متدها است (که مقداری ثابت است) نه تعداد خطوط کد (که با هر پیاده‌سازی ممکن است تغییر کند.) علاوه بر آن، حاشیه‌نویسی در لابه‌لای کدها زمان‌بر بوده و یک منبع خطا نیز به حساب می‌آید [۳۷].

- ایده‌های ارائه شده در این پایان‌نامه، پیاده‌سازی شده و ضمن ارائه تجربیات خود در مورد پیاده‌سازی چنین ابزارهایی، نشان داده‌ایم که دقت الگوریتم‌های پیشنهاد شده، قابل مقایسه با دقیق‌ترین ابزارهای ایستای کشف رقابت است.

بنابراین، سهم ما از حل مشکلات موجود در زمینه طراحی و ساخت ابزارهای تحلیل ایستای کشف رقابت داده مبتنی بر تحلیل‌های جریان داده افزایش مقیاس‌پذیری این‌گونه ابزارها، بدون از دست دادن درستی، دقت، قابلیت آن‌ها در تحلیل برنامه‌های باز، و یا بالا رفتن میزان حاشیه‌نویسی لازم در برنامه‌ها است. برخلاف ابزارهای موجود برای تحلیل برنامه‌های باز (مانند [۲])، روش ارائه شده در این پایان‌نامه، نیازی به ساخت یک برنامه مصنوعی به‌منظور بررسی تمام حالات ممکن برای فراخوانی متدها ندارد، چرا که تحلیل ارائه شده پیمانه‌ای است و قابلیت این را دارد که متدها را جداگانه (و بدون نیاز به برنامه مشتری) تحلیل کند و از این طریق اطلاعات سودمندی را در مورد رخدادهای تولید شده در هر متد و رابطه همروندی بین آن‌ها کسب کند؛ از سویی دیگر با استفاده از حاشیه‌نوشته‌های موجود در صورت هر متد این اطلاعات همروندی دقیق‌تر ساخته می‌شوند.

۳.۱ ساختار پایان‌نامه

فصل بعد، حاوی مرور کوتاهی بر چند مفهوم ریاضی و نیز مفاهیم پایه‌ای تحلیل برنامه‌ها و زبان‌های برنامه‌سازی است. در فصل ۳، مرتبط‌ترین کارها با پژوهش انجام شده در این پایان‌نامه را با جزئیات بیش‌تری مرور کرده‌ایم. فصل ۴، بدنه اصلی راه‌حل ارائه شده در این پایان‌نامه را در بر می‌گیرد. در این فصل، ضمن ارائه یک تحلیل همروندی نوین، یک سیستم کشف رقابت داده مبتنی بر تحلیل یاد شده ارائه می‌شود. معماری پیشنهادی برای پیاده‌سازی سیستم‌هایی که از روش‌های ارائه شده در این پایان‌نامه استفاده می‌کنند نیز مورد بحث قرار می‌گیرد. فصل ۵، مباحث مربوط به پیاده‌سازی را در بر دارد. در این فصل، همچنین، نتایج آزمایش یک پیاده‌سازی از الگوریتم‌های ارائه شده در فصل چهارم ارائه می‌شود. در فصل ۶، تحلیل گراف فراخوانی مورد استفاده در این پایان‌نامه، و نیز یک تحلیل اشاره‌گر پیمانه‌ای ارائه می‌شود. در فصل ۷، پژوهش انجام شده در این پایان‌نامه با مرتبط‌ترین پژوهش‌های انجام شده مقایسه می‌شود. گزارش پایان‌نامه، در فصل ۸، با ارائه یک نتیجه‌گیری کلی و چشم‌اندازی از کارهای آینده خاتمه می‌یابد. در پیوست اول پایان‌نامه، سیستم نوع استاندارد زبان جاوای همروند ارائه شده است. در نهایت، در پیوست دوم درستی الگوریتم کشف رقابت ارائه شده در فصل ۴ اثبات می‌شود.

فصل دوم

پیش‌زمینه

در این فصل مفاهیمی مربوط به مجموعه‌های مرتب جزئی و نظریه مشبک‌ها، نظریه زبان‌ها، سیستم‌های نوع، و تحلیل برنامه‌ها ارائه شده است. دانستن این مفاهیم برای درک بهتر مطالب ارائه شده در این پایان‌نامه ضروری است.لازم به یادآوری است که مطالب این فصل بیش‌تر زنجیره‌ای از تعریف‌های پایه‌ای از مباحث یاد شده است. بنابراین، خوانندگانی که با این مفاهیم آشنایی کامل دارند می‌توانند این فصل را به‌طور سطحی مطالعه کنند.

۱.۲ مجموعه‌های مرتب جزئی

مجموعه‌های مرتب جزئی و مشبک‌ها نقشی اساسی در تحلیل برنامه‌ها دارند. در این بخش ابتدا ترتیب‌های جزئی، مجموعه‌های مرتب جزئی، و خواص آن‌ها را معرفی می‌کنیم. سپس مشبک‌ها، مشبک‌های کامل، و در نهایت قضیه نقطه‌ثابت تارسکی بررسی می‌شود. خوانندگان برای دریافت بحث‌هایی جامع‌تر در زمینه مجموعه‌ها و مجموعه‌های مرتب جزئی می‌توانند به انواع کتاب‌های ریاضیات گسسته مانند [۳۸] یا کتاب‌های مهندسی نرم‌افزار صوری مانند [۳۹] مراجعه کنند. در نهایت مجموعه پیوست‌های کتاب [۱۷] در بر دارنده بحثی جامع در مورد مشبک‌ها و خواص آن‌ها است.

تعریف ۱.۲ (رابطه). دو مجموعه دلخواه A و B مفروض است. هر زیرمجموعه R از $A \times B$ را یک **رابطه** (دو موضعی) از A به B می‌گویند. برای هر a و b، به ترتیب، در مجموعه‌های A و B داریم:

$$a\ R\ b \iff (a, b) \in R,$$

که در آن $a\ R\ b$ به‌صورت «a با b رابطه R دارد» خوانده می‌شود. توجه کنید که $a\ R\ b$ را به‌صورت $R(a, b)$ نیز می‌نویسیم. هر زیر مجموعه از $A \times A$ یک رابطه بر روی A نام دارد. در نهایت لازم به یادآوری است که از این به بعد منظور ما از رابطه همان رابطه دو موضعی است، مگر اینکه تعداد مواضع رابطه به‌صورت صریح بیان شود. ■

تعریف ۲.۲ (ترتیب جزئی). فرض کنید R یک رابطه بر روی مجموعه A باشد. گوییم R یک **ترتیب جزئی** بر روی A است هرگاه دارای خواص زیر باشد:

- بازتابی: $\forall a \in A \cdot a \ R \ a$،
- پادتقارنی: $\forall a_۱, a_۲ \in A \cdot (a_۱ \ R \ a_۲ \land a_۲ \ R \ a_۱) \implies a_۱ = a_۲$، و
- تعدی: $\forall a_۱, a_۲, a_۳ \in A \cdot (a_۱ \ R \ a_۲ \land a_۲ \ R \ a_۳) \implies a_۱ \ R \ a_۳$.

هر رابطه که دارای خواص پادتقارنی و تعدی باشد، اما بازتابی نباشد یک ترتیب جزئی غیر بازتابی نام دارد. ■

تعریف ۳.۲ (مجموعه مرتب جزئی). یک مجموعه، L، به همراه یک ترتیب جزئی $\sqsubseteq$ بر روی آن یک **مجموعه مرتب جزئی** نام دارد و به‌صورت $(L, \sqsubseteq)$ نشان می‌دهند. توجه کنید که در ادامه این پایان‌نامه از $l_۱ \sqsupseteq l_۲$ به جای $l_۱ \sqsubseteq l_۲$ استفاده می‌کنیم. همچنین، از $l_۱ \sqsubset l_۲$ برای نشان دادن $l_۱ \sqsubseteq l_۲$ زمانی که $l_۱ \neq l_۲$ است، استفاده می‌کنیم. ■

در صورتی که مجموعه مرتب جزئی $(L, \sqsubseteq)$ مفروض باشد، برای زیرمجموعه دلخواه Y از L عنصر l از L را یک **کران بالا** برای Y گویند هرگاه $\forall y \in Y \cdot y \sqsubseteq l$. به‌طور مشابه، عنصر l از L را یک **کران پایین** Y گویند هرگاه $\forall y \in Y \cdot y \sqsupseteq l$. **کوچکترین کران بالا** برای مجموعه Y کران بالایی مانند l است که $l \sqsubseteq l$. به قسمی که l یک کران بالای دلخواه برای این مجموعه باشد. همچنین، **بزرگترین کران پایین** برای مجموعه Y کران پایینی مانند l است به‌طوری که برای هر کران پایین دیگر این مجموعه مانند l. داشته باشیم: $l \sqsubseteq l$. توجه کنید که زیرمجموعه‌های Y از یک مجموعه مرتب جزئی الزاما دارای کوچکترین کران بالا یا بزرگترین کران بالا نیستند، اما اگر موجود باشند یکتا هستند و به ترتیب با نمادهای $\bigsqcup Y$ و $\bigsqcap Y$ نشان می‌دهند. عملگر $\bigsqcup$ را عملگر سوپریمم یا پیوند، و عملگر $\bigsqcap$ را عملگر اینفیمم یا اشتراک نیز می‌گویند.

عنصر ماکسیمال ، m، برای زیرمجموعه، Y، از یک مجموعه مرتب جزئی $(L, \sqsubseteq)$ به‌صورت زیر تعریف می‌شود:

- $m \in Y$، و
- $\forall y \in Y \cdot m \sqsubseteq y \implies m = y$.

زیرمجموعه Y از مجموعه مرتب جزئی $(L, \sqsubseteq)$ را یک **زنجیر** گویند هرگاه:

$$\forall y_۱, y_۲ \in Y \cdot (y_۱ \sqsubseteq y_۲) \lor (y_۲ \sqsubseteq y_۱).$$

بنابراین، یک زنجیر یک زیرمجموعه (احتمالاً تهی) از L است که به‌صورت کامل مرتب شده است. بر اساس تعریف زنجیر زنجیرهای صعودی و نزولی به‌صورت زیر قابل تعریف است:

تعریف ۴.۲. یک دنباله $\langle l_i \rangle$ از عناصر L ($l_i \in L$ برای هر $i \in \mathbb{N}$) یک **زنجیر صعودی** است، هرگاه:

$$\forall n, m \in \mathbb{N} \cdot n \leq m \implies l_n \sqsubseteq l_m.$$

به‌طور مشابه دنباله $\langle l_i \rangle$ از عناصر L را یک **زنجیر نزولی** گویند، هرگاه:

$$\forall n, m \in \mathbb{N} \cdot n \leq m \implies l_n \sqsupseteq l_m.$$

■

گوییم یک دنباله $\langle l_i \rangle$ در **نهایت ثبیت می‌شود** هرگاه:

$$\exists n. \in \mathbb{N} \forall n \in \mathbb{N} \cdot n \leq n. \implies l_n = l_{n.}.$$

یک مجموعه مرتب جزئی **شرط زنجیر صعودی** را برآورده می‌کند اگر و تنها اگر تمامی زنجیرهای صعودی آن در نهایت تثبیت شوند. به‌طور مشابه مجموعه یاد شده **شرط زنجیر نزولی** را برآورده می‌کند اگر و تنها اگر تمامی زنجیرهای نزولی آن در نهایت تثبیت شوند.

یکی از قضایای مهم در مورد مجموعه‌های مرتب جزئی لم زورن است که به‌صورت زیر قابل بیان است:

لم ۱.۲ (لم زورن). یک مجموعه مرتب جزئی که هر زنجیر از آن دارای کران بالا باشد، دارای حداقل یک عنصر ماکسیمال است.

اثبات. برای مشاهده اثبات این لم به [۴۰] مراجعه شود. ■

۱.۱.۲ مشبک‌های تام

تعریف ۵.۲ (مشبک تام). یک مجموعه مرتب جزئی مانند $(L, \sqsubseteq)$ را **مشبک تام** گویند، هرگاه هر زیرمجموعه دلخواه از آن دارای کوچکترین کران بالا و بزرگترین کران پایین است. یک مشبک تام را معمولاً با یک چندتایی به‌صورت $(L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ نشان می‌دهند که در آن $\bigsqcap L = \bigsqcup \emptyset = \bot$ و $\bigsqcup L = \bigsqcap \emptyset = \top$ است. مؤلفه‌های $\top$ و $\bot$، به ترتیب بزرگترین و کوچکترین، عناصر L هستند. ■

در این پایان‌نامه چندین مشبک تام معرفی می‌شود که هر کدام کاربردها خاص خود را دارند، اما در هیچ کدام از این مشبک‌ها عملگر اشتراک و نیز بزرگترین عنصر کاربردی ندارد. بنابراین، برای سادگی در نمایش این مشبک‌های را به‌صورت $(L, \sqsubseteq, \bigsqcup, \top)$ نشان می‌دهیم.

لم ۲.۲. برای یک مجوعه مرتب جزئی مانند $(L, \sqsubseteq)$ ادعاهای زیر معادل هستند:

- L یک مشبک تام است،
- هر زیرمجموعه از L دارای کوچکترین کران بالا است،
- هر زیرمجموعه از L دارای بزرگترین کران پایین است.

اثبات. برای مشاهده اثبات این قضیه به پیوست اول کتاب [۱۷] مراجعه شود. ■

نقطه‌ثابت. مفهوم نقطه‌ثابت (توابع) در علوم کامپیوتر نظری کاربردهای فراوانی دارد. برای مثال از آن برای بیان و مطالعه معناشناخت توابعی که به‌صورت بازگشتی تعریف شده‌اند استفاده می‌شود. مفهوم نقطه‌ثابت، همچنین، در حوزه تحلیل برنامه‌ها برای تعریف مفهوم پاسخ یک تحلیل استفاده می‌شود.

در حالت کلی نقطه‌ثابت یک تابع f مقداری مانند l است که (تابع به ازای آن تعریف شده و) $f(l) = l$ است. در این پایان‌نامه توابع مورد علاقه ما توابع یکنوایی هستند که بر روی مشبک‌های تام تعریف شده‌اند. حال اگر فرض کنیم

$(L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ یک مشبک تام است، و $f : L \to L$ یک تابع یکنوا بر روی این مشبک است. نقطه‌ثابت این تابع مقداری مانند $l \in L$ است که $f(l) = l$. مجموعه تمام نقطه‌ثابت‌های تابع یاد شده به‌صورت زیر تعریف می‌شود:

$$Fix(f) = \{l \mid f(l) = l\}.$$

تابع را در نقطه l کاهنده گوییم، هرگاه $f(l) \sqsubseteq l$. مجموعه تمام نقاطی را که f به ازای آن‌ها کاهنده است به‌صورت زیر تعریف می‌کنیم:

$$Red(f) = \{l \mid f(l) \sqsubseteq l\}.$$

تابع را در نقطه l افزاینده گوییم، هرگاه $f(l) \sqsupseteq l$. مجموعه تمام نقاطی را که f به ازای آن‌ها افزاینده است، به‌صورت زیر تعریف می‌کنیم:

$$Ext(f) = \{l \mid f(l) \sqsupseteq l\}.$$

حال از آنجایی که L یک مشبک تام بوده و $Fix(f)$ زیرمجموعه‌ای از آن است، همواره می‌توان ادعا کرد که $Fix(f)$ دارای بزرگترین کران پایین (کوچکترین نقطه‌ثابت) و کوچکترین کران بالا (بزرگترین نقطه‌ثابت) است. کوچکترین نقطه‌ثابت f را با $lfp(f)$، و بزرگترین نقطه‌ثابت آن را با $gfp(f)$ نشان می‌دهیم. این مفاهیم به‌صورت زیر قابل تعریف است:

$$lfp(f) = \bigsqcap Fix(f),$$
$$gfp(f) = \bigsqcup Fix(f).$$

قضیه زیر، که به **قضیه نقطه‌ثابت تارسکی** شهرت دارد، تضمین می‌کند که یک تابع یکنوا بر روی یک مشبک تام دارای نقطه‌ثابت است.

قضیه ۳.۲ (نقطه‌ثابت تارسکی). با فرض $(L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ به عنوان یک مشبک تام، در صورتی که $f : L \to L$ یک تابع یکنوا بر روی این مشبک باشد، داریم:

$$lfp(f) = \bigsqcap Red(f) \in Fix(f),$$
$$gfp(f) = \bigsqcup Ext(f) \in Fix(f).$$

اثبات. برای مشاهده اثبات این قضیه به [۱۷] مراجعه شود. ■

۲.۲ نظریه زبان‌ها: بستارهای کِلنه و الحاق

در این بخش نگاهی گذرا به مفهوم بستار خواهیم داشت. هرچند که مفاهیم ارائه شده برای مطالعه سایر بخش‌های پایان‌نامه کافی است، خوانندگان برای دریافت اطلاعات در مورد گرامرهای مستقل از متن می‌توانند به کتاب [۴۱] یا هر کتاب دیگر در زمینه نظریه زبان‌های صوری مراجعه کنند.

در صورتی که S مجموعه‌ای از نمادها باشد (نه الزاماً متناهی) S^* را **بستار ستاره‌ای** مجموعه یاد شده می‌گویند، و به عنوان مجموعه تمام دنباله‌های متشکل از عناصر S تعریف می‌شود. به‌طور مشابه S^+ را **بستار مثبت** مجموعه S می‌گویند، و به‌صورت مجموعه تمام دنباله‌های با طول بیشتر از صفر از عناصر موجود در S تعریف می‌گردد. عناصر موجود در یک بستار را **رشته** می‌گویند. رشته با طول صفر را رشته تهی نامیده و با Λ نشان می‌دهند.

آخرین مطلبی که در رابطه با نظریه زبان‌ها در این پایان‌نامه استفاده کرده‌ایم، مفهوم الحاق دو رشته است. در صورتی که $s_۱$ و $s_۲$ دو رشته از نمادها باشند الحاق این دو رشته را با $s_۱.s_۲$ نشان می‌دهیم و به‌صورت زیر تعریف می‌کنیم:

$$\begin{aligned} s_۱ &= \langle a_۱, \ldots, a_n \rangle \qquad ; n \geq ۰, \\ s_۲ &= \langle b_۱, \ldots, b_m \rangle \qquad ; m \geq ۰. \\ s_۱.s_۲ &= \langle a_۱, \ldots, a_n, b_۱, \ldots, b_m \rangle . \end{aligned}$$

واضح است که عملگر الحاق رشته دارای خاصیت جابجایی نیست.

۳.۲ سیستم‌های نوع

در این بخش نگاهی گذرا به مفهوم نوع و سیستم‌های نوع خواهیم داشت. معرفی هر یک از مفاهیم نوع یا سیستم‌های نوع، همانند سایر مفاهیم ارائه شده در این فصل، نیازمند بحث‌های مفصلی بوده که بدیهی است که در این چند صفحه قادر به بیان آن‌ها نخواهیم بود. بر خلاف سایر مطالب ارائه شده در این فصل یک معرفی سطحی مفهوم سیستم‌های نوع برای درک کامل موارد استفاده از آن در این پایان‌نامه کافی نیست. بنابراین، انتظار می‌رود که خواننده یک آشنایی قبلی با مفهوم نوع، سیستم‌های نوع، و مفاهیم مرتبط مانند حساب λ داشته باشد. برای درک بهتر مفهوم نوع مقاله [۲۶] را پیشنهاد می‌کنیم. مقاله [۲۵] بحث جامعی در مورد سیستم‌های نوع دارد. در نهایت کتاب [۲۷] ضمن این‌که بیشتر مطالب ارائه شده در دو مقاله یاد شده را در بر گرفته است، شامل مباحث کاربردی نوع‌ها در زبان‌های برنامه‌سازی و مقدمه‌ای بر بحث‌های پیش‌رفته‌تر نیز است.

ترجیح می‌دهیم که به جای اینکه به‌طور مستقیم به پاسخ دادن به سوال «نوع چیست؟» بپردازیم، از معرفی مواردی که به نوع‌ها در زبان‌های برنامه‌سازی احتیاج پیدا می‌شود شروع کنیم. برای مثال دنیاهای بدون نوع زیر را تصور کنید:

- رشته‌های بیتی در حافظه اصلی یک کامپیوتر،
- عبارت‌های λ در حساب λ.

دنیای رشته‌های بیتی ملموس‌تر از دنیای دیگر است، بنابراین ابتدا به توضیح دنیای (بدون نوع) رشته‌های بیتی می‌پردازیم. در اینجا منظور ما از بدون نوع این است که فقط یک نوع وجود دارد و آن هم کلمه حافظه (رشته‌های بیتی با اندازه ثابت) است. این دنیا بدون نوع است، چرا که هرچیز (کاراکترها، اعداد، اشاره‌گرها، برنامه‌ها، و غیره) در نهایت باید با استفاده از رشته‌های بیتی نمایش داده شوند. هنگامی که به یک تکه از حافظه (تعدادی بیت) نگاه کنیم، قادر نخواهیم بود که تشخیص دهیم این رشته در اصل چه چیزی را نمایش می‌دهد. تنها راه دانستن معنای آن تکه از حافظه مراجعه به یک تفسیر خارجی از محتوای آن است.

در حساب λ هر چیزی با استفاده از یک تابع نمایش داده می‌شود (یا اینکه هدف این است که آن چیز یک تابع را نشان دهد). اعداد، عملگرها بر روی اعداد، ساختارهای شرطی و تکرار، رشته‌های بیتی، و غیره را می‌توان با استفاده از توابع مناسبی نمایش داد. در اینجا نیز تنها یک نوع وجود دارد: نوع توابع از مجموعه مقادیر به مجموعه مقادیر به قسمی که مقادیر خود توابعی از همان نوع هستند.

وقتی که وارد یک دنیای بدون نوع می‌شویم، برای شروع کار در این دنیا شروع به دسته‌بندی اشیاء به‌صورت‌های مختلف و برای مقاصد مختلف می‌کنیم. نوع‌ها، به زبان ساده، در هر فضایی برای دسته‌بندی اشیاء (آن فضا) بر حسب رفتار و کاربرد معرفی می‌شوند. دسته‌بندی اشیاء بر اساس اهداف کاربرد آن‌ها، درنهایت، منجر به توسعه سیستم‌های نوع می‌شود. بنابراین، نوع‌ها به‌طور طبیعی با دسته‌بندی و تجرید در فضاهای بدون نوع معرفی می‌شوند. در حافظه کامپیوتر بین کاراکترها، اعداد، و برنامه‌های ذخیره شده تفاوت قائل می‌شویم. در حساب λ برخی از توابع برای نشان دادن مقادیر بولی، برخی دیگر برای نشان دادن اعداد، و غیره به کار می‌روند.

دنیاهای بدون نوع اشیاء محاسباتی، به‌طور طبیعی، به زیرمجموعه‌هایی با رفتارهای همسان افزار می‌شوند. نوع‌ها شناسه‌هایی برای مجموعه‌های اشیاء با رفتار همسان است. برای مثال، رفتار همسان مجموعه اعداد طبیعی این است که همه آن‌ها دارای مجموعه عملگرهای مجاز برای استفاده یکسانی دارند. رفتار همسان توابع از مجموعه بولی به مجموعه اعداد طبیعی نیز این است که همه آن‌ها با دریافت یک عملوند از نوع بولی یک نتیجه از نوع طبیعی محاسبه می‌کنند.

بعد از انجام دسته‌بندی متوجه می‌شویم که یک چنین دسته‌بندی موجودیت خارجی ندارد و صرفاً یک دسته‌بندی خیالی است. برای مثال، طبیعی است که سوالاتی مانند: نتیجه «یا» منطقی یک کاراکتر با نمایش بیتی یک دستورالعمل چیست؟ نتیجه اعمال یک عبارت شرطی بر روی یک عدد صحیح چیست؟ به وجود می‌آید. علت به وجود آمدن چنین سوالاتی این است که بعد از انجام دسته‌بندی از هیچ سیستم‌نوعی برای تصمیم در مورد صحیح بودن حالات مختلف ترکیب اشیاء و عملگرها استفاده نشده است. در حقیقت یک هدف اساسی از سیستم‌های نوع اجتناب از مطرح شدن چنین سوالات دست و پا گیر درباره نمایش اشیاء است، که دستیابی به چنین هدفی با جلوگیری از به وجود آمدن وضعیت‌هایی که در آن چنین سوالاتی مطرح می‌گردد صورت می‌گیرد. در ریاضیات نیز، همانند برنامه‌سازی، نوع‌ها محدودیت‌هایی را برای کمک به اعمال درستی وضع می‌کند. برخی از فضاهای بدون نوع مانند نظریه طبیعی مجموعه‌ها از لحاظ منطقی ناسازگار هستند (یعنی ممکن است اثباتی برای گزاره‌ها نادرست نیز وجود داشته باشد). نظریه نوع‌ها با فراهم کردن نسخه‌های نوع‌دار نظریه مجموعه‌ها این ناسازگاری‌ها را به‌طور کامل رفع می‌کند. نسخه‌های نوع‌دار نظریه مجموعه‌ها با وضع محدودیت‌هایی بر روی شیوه تعامل اشیاء از نوع‌های مختلف با هم‌دیگر از تعاملات ناسازگار جلوگیری به عمل می‌آورند.

هدف اصلی سیستم‌های نوع را می‌توان به این صورت بیان کرد: پیش‌گیری از وقوع خطاهای در زمان اجرا با وارسی ایستای (متن) برنامه‌ها. همان‌طور که مشاهده می‌شود، این تعریف از هدف سیستم‌های نوع در اصل یک جمع‌بندی از تعاریف حسی از این موضوع که پیش‌تر ارائه دادیم است. با این حساب می‌توان یک سیستم نوع را به‌صورت زیر تعریف کرد:

یک سیستم نوع یک رویه نحوی برای اثبات نبود رفتارهای خاصی در برنامه‌ها است، که این تضمین با دسته‌بندی عبارات بر اساس نوع مقادیری که هر عبارت به آن ارزیابی می‌شود انجام می‌پذیرد.

این تعریف نیاز به قدری بررسی و تأمل دارد. به منظور سرعت در انتقال مفهوم این موارد را به‌صورت یک فهرست بیان می‌کنیم:

۱- در این تعریف سیستم نوع به عنوان یک رویه معرفی شده است. این یعنی یک سیستم نوع باید محاسبه‌پذیر باشد.

۲- یک سیستم نوع یک رویه نحوی است: در اینجا به طبیعت ایستای سیستم‌های نوع به‌صورت صریح بیان می‌شود. به عبارتی دیگر یک سیستم نوع در اصل یک الگوریتم با ورودی درخت نحوی مجرد است.

۳- در این تعریف منظور از اثبات نبودن برخی از رفتارها در برنامه‌ها این است که سیستم‌های نوع تضمین می‌کنند که در زمان اجرا برخی از رفتارها (مثلاً برخی از خطاها) اتفاق نیفتد. این تضمین خود یک نتیجه دیگر دارد، و آن هم این است که سیستم نوع باید درست باشد: هنگامی که اعلام می‌کند برنامه‌ای خالی از یک رفتار خاص است، واقعاً هیچ اجرایی از آن برنامه موجود نباشد که در آن رفتار (نامطلوب) مورد نظر اتفاق بیفتد.

۴- در نهایت آن مفهومی که از این تعریف به ذهن خواننده منتقل می‌شود، این است که اثبات عدم وجود رفتارهای خاص در یک برنامه با انتساب نوع به هر عبارت (یا دستورالعمل، و یا جمله در حالت کلی) صورت می‌گیرد. پیش‌تر حسی از این دسته‌بندی و یک چنین تضمینی در نسخه‌های نوع‌دار نظریه مجموعه‌ها ارائه شد. وجه تسمیه سیستم‌های نوع نیز همین شیوه دسته‌بندی آن‌ها است.

همان‌طور که از تعریف بالا بر می‌آید سیستم‌های نوع باید به‌صورت دقیق توصیف شوند تا استدلال در مورد درستی آن‌ها امکان‌پذیر باشد. سیستم‌های نوع را می‌توان به‌صورت استقرایی در قالب یک سیستم اثبات توصیف کرد. در این شیوه از توصیف اثبات درستی (به دلیل استقرایی بودن سیستم) بسیار ساده‌تر می‌شود. همچنین، درک یک تعریف استقرایی که متشکل از قواعد استنتاج و اصول است بسیار ساده‌تر از یک الگوریتم غیر استقرایی (به همراه جزئیات غیرضروری احتمالاً) پیچیده است.

برای روش‌تر شدن این موضوع زبان ساده NB-Fun را در نظر بگیرید: در این زبان دو نوع شئ اعداد طبیعی و مقادیر بولی وجود دارد، به‌طوری که عملگر $+$ برای جمع دو عدد صحیح، عملگر $\wedge$ برای محاسبه ترکیب وصلی دو مقدار بولی، ساختار fun برای تعریف یک تابع، و در نهایت ساختاری برای اعمال توابع تعریف شده بر روی عملوندها موجود است. بدیهی است که جمع دو مقدار بولی و یا ترکیب وصلی دو عدد طبیعی بی‌معنا است (البته باید به معنا شناخت این زبان مراجعه کنیم که در اینجا برای سادگی به یک معناشناخت زبانی و غیرصوری بسنده می‌کنیم). نحو مجرد برای زبان NB-Fun به شرح زیر است:

$$e \quad ::= \quad n \mid b \mid e + e \mid e \wedge e \mid \text{fun } x \text{ in } e \mid e\ e \mid (e)$$

$$\begin{aligned} n &\in \{۰, ۱, ۲, \dots\} \\ b &\in \{\text{true}, \text{false}\} \end{aligned}$$

در این زبان هیچ چیز مانع نوشتن چنین برنامه بی‌معنایی نمی‌شود:

$$(\text{fun } x \text{ in } x + ۱)\ \text{true}.$$

ایده‌ای که برای جلوگیری از نوشتن چنین برنامه‌هایی می‌تواند مفید واقع شود، این است که به هر دسته از عبارت‌های برنامه‌ها نوعی منتسب کنیم. برای مثال به نظر می‌رسد بهتر است نوع Nat را برای اعداد طبیعی انتخاب کنیم، و نوع Bool را برای مقادیر بولی به‌کار بریم. همچنین، به منظور انتساب نوع برای توابع (برای پرهیز از پیچیدگی) یک حاشیه‌نویسی در

صورت تابع معرفی می‌کنیم:

$$e \quad ::= \quad \cdots \mid \text{fun } x : t \text{ in } e \mid \ldots$$

$$\vdots$$

$$t \quad ::= \quad \mathsf{Nat} \mid \mathsf{Bool} \mid t \to t$$

همان‌طور که مشاهده می‌کنیم، این‌بار برنامه‌نویس موظف به حاشیه‌نویسی پارامتر توابع برای تصریح نوع (آرگومان) مورد انتظار تابع است. توجه کنید که این حاشیه‌نویسی را می‌توان به عنوان یک توصیف نیازمندی در نوع آرگومان تابع پاس شده به تابع دانست (به این ترتیب سیستم نوع وظیفه صحت‌سنجی برنامه‌ها برای تبعیت از این توصیف نیازمندی را دارد.) حال می‌توان سیستم نوعی برای زبان NB-Fun توصیف کرد. این سیستم نوع گزاره‌هایی به فرم $E \vdash e : t$ را اثبات می‌کند، که در آن E محیط انتساب نوع است که برای انتساب نوع به متغیرهای آزاد در یک عبارت (متغیر xای که در یک عبارت به شکل $\text{fun } x : t \text{ in } e$ ظاهر نشده باشد) به کار میرود. محیط انتساب نوع را می‌توان به‌صورت زیر تعریف کرد:

$$E ::= \emptyset \mid E, x : t$$

که بیان می‌کند: محیط انتسب نوع یا تهی است، یا از افزودن یک مورد $x : t$ به یک محیط انتساب نوع به دست می‌آید. یک محیط انتساب نوع را خوش‌فرم گویند هرگاه: (۱) تهی باشد، یا (۲) برابر با $E, x : t$ باشد، که در آن E خوش‌فرم بوده و x عضو مجموعه متغیرهای موجود در E نیست. این موضوع را با نماد $WF(E)$ نشان می‌دهیم. حال سیستم نوع را به‌صورت استقرایی زیر تعریف می‌کنیم:

$$\frac{E = E_1, x : t, E_2 \qquad WF(E)}{E \vdash x : t}\ (\text{Rule} - \text{Var}) \qquad \frac{n \in \{0, 1, 2, \ldots\} \qquad WF(E)}{E \vdash n : \mathsf{Nat}}\ (\text{Rule} - \text{Nat}) \qquad \frac{b \in \{\text{true}, \text{false}\} \qquad WF(E)}{E \vdash b : \mathsf{Bool}}\ (\text{Rule} - \text{Bool})$$

$$\frac{E \vdash e_1 : \mathsf{Nat} \qquad E \vdash e_2 : \mathsf{Nat}}{E \vdash e_1 + e_2 : \mathsf{Nat}}\ (\text{Rule} - \text{Plus}) \qquad \frac{E \vdash e_1 : \mathsf{Bool} \qquad E \vdash e_2 : \mathsf{Bool}}{E \vdash e_1 \wedge e_2 : \mathsf{Bool}}\ (\text{Rule} - \text{And})$$

$$\frac{E, x : t \vdash e : t'}{E \vdash \text{fun } x : t \text{ in } e : t \to t'}\ (\text{Rule} - \text{Fun}) \qquad \frac{E \vdash e_1 : t \to t' \qquad E \vdash e_2 : t}{E \vdash e_1\ e_2 : t'}\ (\text{Rule} - \text{App})$$

قواعد استنتاج و اصول خود توصیفی هستند و نیازی به موشکافی آن‌ها احساس نمی‌شود. با توسعه یک معناشناخت صوری برای یک زبان می‌توان درستی این سیستم نوع را نیز اثبات کرد. اما بحث اثبات درستی سیستم‌های نوع فراتر از حوزه بحث ما در این فصل است. خوانندگان علاقه‌مند می‌توانند به کتاب [۲۷] برای مشاهده اثبات درستی برای انواع سیستم‌های نوع با پیچیدگی‌های مختلف مراجعه کنند.

۴.۲ تحلیل ایستای برنامه‌ها

تحلیل ایستای یک برنامه به روشی، در زمان کامپایل، گفته می‌شود که برای تخمین ایمن رفتار و مقادیری که در زمان اجرای آن برنامه مشاهده می‌گردد به‌کار گرفته می‌شود. همان‌طور که از این تعریف بر می‌آید، تحلیل ایستای یک برنامه نیازی به هیچ اجرا از آن برنامه را ندارد، و اطلاعات مربوط به رفتار برنامه مورد نظر صرفاً با توجه به متن برنامه استخراج می‌شود. این گونه تحلیل‌ها در مقابل تحلیل‌های پویا قرار دارند، چرا که تحلیل‌های پویا رفتار برنامه‌ها را حین اینکه اجرا می‌شوند مورد تحلیل قرار می‌دهد.

تحلیل ایستای برنامه‌ها، همانند نوع‌ها و سیستم‌های نوع، مبحث گسترده‌ای را در حوزه علوم کامپیوتر نظری به خود اختصاص داده است. در این بخش قصد داریم مروری کوتاه بر مهم‌ترین مفاهیم تحلیل برنامه‌ها که در این پایان‌نامه مورد استفاده قرار گرفته است داشته باشیم. توجه کنید که فرض ما بر این است که خواننده یک آشنایی قبلی با مفاهیم تحلیل ایستای برنامه‌ها (مخصوصاً تحلیل‌های مبتنی بر جریان داده و تحلیل‌های مبتنی بر قید) دارد، ما کتاب [۱۷] را به یک مرجع جامع آموزشی در زمینه تحلیل ایستای برنامه‌ها معرفی می‌کنیم. برخی از کتاب‌های کامپایلر نیز حاوی مطالبی در مورد بهینه‌سازی با استفاده از تحلیل‌های مبتنی بر جریان داده است، که می‌توانند نقطه شروعی برای بحث تحلیل ایستا باشند.

تحلیل ایستای برنامه‌ها به چندین نوع دسته‌بندی می‌شود. اولین و قدیمی‌ترین آن‌ها **تحلیل جریان داده** است. تحلیل جریان داده سعی در شناسایی (مثلاً از طریق محاسبه) مجموعه مقادیر ممکن در هر نقطه از برنامه است. از آنجایی که نمایش غالب برای برنامه‌ها در تحلیل جریان داده گراف جریان کنترل است، این تحلیل در اصل مجموعه مقادیر ممکن را در هر نقطه از برنامه (که متناظر با یک گره از گراف جریان کنترل است) تعیین می‌کند. در این تحلیل ابتدا با تنظیم معادلات جریان داده در هر گره از گراف جریان کنترل یک دستگاه معادلات جریان داده برای کل گراف جریان کنترل ساخته می‌شود. سپس با استفاده از یک الگوریتم تکراری محاسبه نقطه‌ثابت دستگاه یاد شده را حل می‌کنند. لازم به یادآوری است که یک تحلیل الزاماً نیازی به محاسبه جواب برای دستگاه یادشده ندارد، و در برخی از موارد ارائه یک اثبات برای وجود جواب کفایت می‌کند. در صورتی که بخواهیم یک الگوریتم برای محاسبه دستگاه معادلات ارائه دهیم نیاز داریم که اطلاعاتی قرار است از متن یک برنامه استخراج شود از یک مشبک تام با شرط زنجیر صعودی باشند (در غیر این‌صورت تضمینی برای خاتمه رویه وجود ندارد). در صورتی هم که نیازی به محاسبه جواب نداریم، برای اثبات وجود یک جواب نیازی به برآورده کردن شرط زنجیر صعودی برای مشبک تام یاد شده نیست.

نوع دیگری از تحلیل برنامه‌ها **تحلیل مبتنی بر قید** است. در این شیوه از تحلیل ابتدا یک دستگاه از قیود برای برنامه ساخته می‌شود. این دستگاه سپس توسط یک حل کننده قیود حل می‌شود. مرسوم‌ترن نوع قیود، قیدهایی هستند که به‌صورت تساوی یک شناسه با یک مقدار ارائه می‌شوند. چنین دستگاه قیودی (با استفاده از یک ساختار داده اجتماع‌جستجو) در زمانی تقریبا خطی حل می‌شود. دستگاه‌های قیود با قیدهایی که در آن‌ها به‌جای تساوی از شمول استفاده شده است، زمانی از مرتبه $O(n^۳)$ برای حل شدن نیاز دارند (که در آن n تعداد قیود در دستگاه است.) روش‌های مبتنی بر قید به‌طور عمده مستقل از جریان هستند، چرا که این‌گونه تحلیل‌ها جریان کنترل برنامه‌ها را در نظر نمی‌گیرند. همچنین، تحلیل‌های مبتنی بر قید معمولاً نیاز به در نظر گرفتن کل برنامه را دارند.

تفسیر مجرد نوعی دیگر از تحلیل ایستای برنامه‌ها است. تفسیر مجرد اطلاعات در مورد معنای برنامه‌ها را با اجرای نمادین آن‌ها به دست می‌آورد. این شیوه از تحلیل عمومی‌تر از دو نوع دیگر از تحلیل‌های ایستا است، چرا که می‌توان آن‌ها برحسب تفسیر مجرد بیان کرد. تفسیر مجرد بسیار دقیق و در عین حال بسیار پرهزینه است، به‌طوری که تا کنون صرفاً

برای تحلیل تکه‌های کوچک برنامه‌ها مورد استفاده قرار گرفته است.

علاوه بر دسته‌بندی بالا تحلیل‌های ایستا را بر اساس حساسیت به جریان و حساسیت به زمینه نیز دسته‌بندی می‌کنند. یک تحلیل **مستقل از جریان** در استخراج اطلاعات در مورد رفتار برنامه‌ها ترتیب دستورالعمل‌ها را در نظر نمی‌گیرد. در مقابل پاسخ تحلیل‌های **حساس به جریان** با تغییر ترتیب دستورالعمل‌های برنامه ممکن است تغییر کند. به عبارتی دیگر تحلیل‌های مستقل از جریان یک جواب برای کل برنامه تعیین می‌کنند، حال آنکه تحلیل‌های حساس به جریان یک جواب مجزا برای هر نقطه از برنامه تعیین می‌کنند. تحلیل‌های مستقل از جریان، به‌طور معمول، کاراتر اما غیر دقیق‌تر از تحلیل‌های حساس به جریان هستند. حساسیت به زمینه به این موضوع مربوط می‌شود که آیا تحلیل بین فراخوانی‌های مختلف یک متد یا یک رویه تفاوت قائل می‌شود یا خیر. همان‌طور که می‌دانیم یک رویه در یک برنامه در زمینه‌های مختلف (بر حسب مقادیر متغیرهای عمومی و پارامترهای پاس شده به رویه) می‌تواند فراخوانی شود. در یک تحلیل **مستقل از زمینه** اطلاعات بدست آمده از هر یک از زمینه‌های فراخوانی با هم ادغام (از طریق عملگر کوچکترین کران بالا) می‌شوند. این در حالی است که در تحلیل‌های **حساس به زمینه** اطلاعات زمینه‌های مختلف از هم تفکیک می‌شوند. بدیهی است که یک تحلیل حساس به زمینه دقیق‌تر و پرهزینه‌تر از یک تحلیل مستقل از زمینه خواهد بود.

برای آشنایی بیش‌تر با تحلیل جریان داده، در ادامه، یک مثال از تحلیل جریان داده یک برنامه ارائه می‌دهیم. قبل از هر چیز نیاز به تعریف یک زبان داریم. یک برنامه در این زبان یک دستورالعمل است که خود می‌تواند دنباله‌ای از دستورالعمل‌ها باشد. فرض می‌کنیم دستورالعمل‌های برنامه مورد نظر برچسب دار هستند. به هر دستورالعمل برچسب دار یک بلوک ابتدایی گفته می‌شود. همچنین، فرض می‌کنیم که هر دستورالعمل در یک برنامه دارای برچسب منحصر به فرد است.

از متغیرهای نحوی A، B، و S به ترتیب برای نشان دادن عبارت‌های محاسباتی، عبارت‌های بولی، و دستورالعمل‌ها استفاده می‌کنیم. همچنین، از متغیرهای نحوی X، N، و ℓ به ترتیب برای نشان دادن شناسه‌های متغیرها، اعداد طبیعی، و برچسب‌ها استفاده می‌کنیم. در نهایت از متغیرهای نحوی O_a، O_b، و O_r برای نشان دادن عملگرهای محاسباتی، عملگرهای بولی، و عملگرهای رابطه‌ای استفاده می‌کنیم. به این ترتیب می‌توان نحو زبان مورد نظر را به‌صورت زیر تعریف کرد:

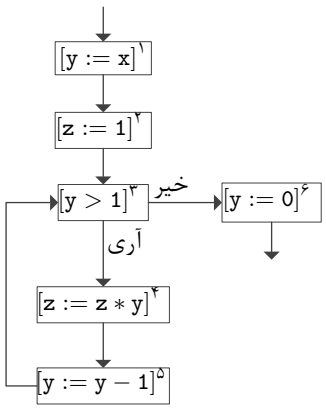
$$
\begin{aligned}
A &::= X \mid N \mid A_1\ O_a\ A_1 \\
B &::= \texttt{true} \mid \texttt{false} \mid \texttt{not}\ B \mid B_1\ O_b\ B_2 \mid A_1\ O_r\ A_2 \\
S &::= [X{:=}A]^{\ell} \mid S_1; S_2 \mid \texttt{while}\ [B]^{\ell}\ \texttt{do}\ S
\end{aligned}
$$

در ادامه مجموعه تمام رشته‌هایی که توسط متغیر نحوی ℓ تولید می‌شوند (مجموعه تمام برچسب‌ها) را با Lab نشان می‌دهیم، و برای سادگی فرض می‌کنیم که این مجموعه برابر با مجموعه اعداد طبیعی است.

برنامه زیر فاکتوریل عددی را که در متغیر `x` ذخیره شده است را محاسبه کرده، و نتیجه را به `z` منتسب می‌کند. توجه کنید که در این مثال از نمادهای پرانتز برای رفع ابهام استفاده شده است.

$$
[\texttt{y} =: \texttt{x}]^1\,;\,[\texttt{z} := 1]^2\,;\,\texttt{while}\ [\texttt{y} > 1]^3\ \texttt{do}\ \Big([\texttt{z} := \texttt{z} * \texttt{y}]^4\,;\,[\texttt{y} := \texttt{y} - 1]^5\Big)\,;\,[\texttt{y} := 0]^6 \qquad (۱.۲)
$$

همان‌طور که پیش‌تر نیز اشاره کردیم در تحلیل جریان داده برای یک برنامه آن را با استفاده از یک گراف به نام گراف جریان کنترل نشان می‌دهند. گره‌های گراف جریان کنترل بلوک‌های ابتدایی و کمان‌های آن نشان دهنده چگونگی انتقال جریان کنترل بین بلوک‌های ابتدایی است. در زبان ارائه شده در این بخش انتقال جریان کنترل توسط ساختارهای ; و `while` اعمال



شکل ۱.۲: گراف جریان کنترل برای برنامه ۱.۲

می‌شود. توجه کنید که همواره فرض می‌کنیم یک بلوک ابتدایی مانند یک تابع انتقال حالت بوده که با دریافت حالت جاری سیستم به عنوان ورودی، حالتی که سیستم با اجرای بلوک مورد نظر به آن منتقل می‌شود را به عنوان خروجی تولید می‌کند. بنابراین، برای یک بلوک ابتدایی یک یا چند نقطه ورود و یک نقطه خروج در نظر می‌گیریم. با توجه به این توضیحات به راحتی می‌توان مشاهده کرد که گراف جریان کنترل برای برنامه ۱.۲ به‌صورت گراف نشان داده شده در شکل ۱.۲ است.

تحلیلی که در این بخش می‌خواهیم به عنوان مثالی از یک تحلیل جریان داده ارائه دهیم تحلیل تعاریف دستیابی یا همان RDA است. قبل از اینکه توضیح دهیم که تحلیل RDA چیست، نیاز داریم مفهوم رسیدن یک انتساب به یک نقطه از برنامه را معرفی کنیم. یک انتساب به فرم $[X:=A]^{\ell}$ به یک نقطه از برنامه (یک بلوک ابتدایی) می‌رسد، هرگاه یک اجرا از آن برنامه تا نقطه یاد شده از برنامه وجود داشته باشد که در آن آخرین بار در نقطه ℓ به متغیر X مقداری منتسب شده است. تحلیل RDA برای هر نقطه از برنامه تعیین می‌کند که چه انتساب‌هایی ممکن است به آن نقطه برسند. برای مثال، در برنامه ۱.۲ انتساب $[\mathrm{y} =: \mathrm{x}]^{۱}$ به ورودی بلوک $[\mathrm{z} := 1]^{۲}$ می‌رسد؛ به منظور سادگی در نمایش این حقیقت را با گفتن $(\mathrm{y}, ۱)$ (یعنی انتساب متغیر y در نقطه ۱) به نقطه ۲ می‌رسد بیان می‌کنیم. همچنین، می‌گوییم $(\mathrm{x}, ?)$ به نقطه ۲ می‌رسد، که در آن منظور ما از $?$ برچسب ویژه است که در هیچ برنامه‌ای ظاهر نشده است. از این نماد برای نشان دادن مکان آخرین انتساب برای متغیرهایی که به‌صورت صریح مقداردهی اولیه نشده‌اند استفاده می‌کنیم. اصلی‌ترین کاربرد تحلیل RDA در بحث بهینه‌سازی کامپایلرها است، که برای حذف کدهای مرده به‌کار می‌رود.

با قدری تأمل متوجه می‌شویم که اطلاعات ارائه شده در جدول ۱.۲ همان اطلاعات RDA مورد انتظار و آرمانی برای هر نقطه از برنامه ۱.۲ است.

انجام یک تحلیل جریان داده برای تعیین اطلاعات RDA به وسیله استخراج معادلات جریان داده از متن برنامه صورت می‌گیرد. در حالت کلی دو نوع معادله جریان داده وجود دارد: دسته اول اطلاعات خروجی یک گره را به اطلاعات ورودی همان گره مرتبط می‌کنند، و دسته دوم معادلات جریان داده اطلاعات ورودی گره‌ها را به اطلاعات خروجی تمام گره‌هایی مرتبط می‌کند که جریانی از آن‌ها به گره مورد نظر وارد می‌شود. به عبارتی بهتر گروه اول معادلات اطلاعات خروجی یک گره را برحسب اطلاعات ورودی همان گره بیان می‌کند، و گروه دوم معادلات اطلاعات ورودی یک گره را بر حسب اطلاعات

جدول ۱.۲: اطلاعات RDA برای هر نقطه از برنامه ۱.۲

$RD_{exit}(\ell)$	$RD_{entry}(\ell)$	ℓ
$(\mathtt{x},?),(\mathtt{y},۱),(\mathtt{z},?)$	$(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)$	۱
$(\mathtt{x},?),(\mathtt{y},۱),(\mathtt{z},۲)$	$(\mathtt{x},?),(\mathtt{y},۱),(\mathtt{z},?)$	۲
$(\mathtt{x},?),(\mathtt{y},۱),(\mathtt{y},۵),(\mathtt{z},۲),(\mathtt{z},۴)$	$(\mathtt{x},?),(\mathtt{y},۱),(\mathtt{y},۵),(\mathtt{z},۲),(\mathtt{z},۴)$	۳
$(\mathtt{x},?),(\mathtt{y},۱),(\mathtt{y},۵),(\mathtt{z},۴)$	$(\mathtt{x},?),(\mathtt{y},۱),(\mathtt{y},۵),(\mathtt{z},۲),(\mathtt{z},۴)$	۴
$(\mathtt{x},?),(\mathtt{y},۵),(\mathtt{z},۴)$	$(\mathtt{x},?),(\mathtt{y},۱),(\mathtt{y},۵),(\mathtt{z},۴)$	۵
$(\mathtt{x},?),(\mathtt{y},۶),(\mathtt{z},۲),(\mathtt{z},۴)$	$(\mathtt{x},?),(\mathtt{y},۱),(\mathtt{y},۵),(\mathtt{z},۲),(\mathtt{z},۴)$	۶

خروجی تمام گره‌هایی بیان می‌کند که جریانی از آن‌ها خارج شده و به گره مورد وارد می‌شود. برای گراف جریان کنترل نمایش داده شده در شکل ۱.۲، دسته اول معادلات را به‌صورت زیر بیان می‌کنیم:

$$
\begin{aligned}
RD_{exit}(۱) &= (RD_{entry}(۱) - \{(\mathtt{y},\ell) \mid \ell \in Lab\}) \cup \{(\mathtt{y},۱)\},\\
RD_{exit}(۲) &= (RD_{entry}(۲) - \{(\mathtt{z},\ell) \mid \ell \in Lab\}) \cup \{(\mathtt{z},۲)\},\\
RD_{exit}(۳) &= RD_{entry}(۳),\\
RD_{exit}(۴) &= (RD_{entry}(۴) - \{(\mathtt{z},\ell) \mid \ell \in Lab\}) \cup \{(\mathtt{z},۴)\},\\
RD_{exit}(۵) &= (RD_{entry}(۵) - \{(\mathtt{y},\ell) \mid \ell \in Lab\}) \cup \{(\mathtt{y},۵)\},\\
RD_{exit}(۶) &= (RD_{entry}(۶) - \{(\mathtt{y},\ell) \mid \ell \in Lab\}) \cup \{(\mathtt{y},۶)\}.
\end{aligned}
$$

همان‌طور که مشاهده می‌شود معادلات به این شکل نوشته شده‌اند که برای یک مورد دستور انتساب به فرم $[X:=A]^{\ell'}$ تمام زوج‌های مرتب به شکل (X,ℓ) را از $RD_{entry}(\ell')$ حذف کرده و (X,ℓ') را برای بدست آوردن $RD_{exit}(\ell)$ به نتیجه اضافه می‌کنیم. برای سایر بلوک‌های ابتدایی به‌فرم کلی $[\ldots]^{\ell'}$ نیز $RD_{entry}(\ell')$ را برابر با $RD_{exit}(\ell)$ قرار می‌دهیم، چرا که سایر بلوک‌ها، از دیدگاه تحلیل RDA، هیچ اثری در محاسبه اطلاعات RDA ندارد.

دسته دوم معادلات برای این مثال را به‌صورت زیر تعریف می‌کنیم:

$$
\begin{aligned}
RD_{entry}(۲) &= RD_{exit}(۱),\\
RD_{entry}(۳) &= RD_{exit}(۲) \cup RD_{exit}(۵),\\
RD_{entry}(۴) &= RD_{exit}(۳),\\
RD_{entry}(۵) &= RD_{exit}(۴),\\
RD_{entry}(۶) &= RD_{exit}(۳).
\end{aligned}
$$

در حالت کلی معادلاتی به فرم $RD_{entry}(\ell) = RD_{exit}(\ell_۱) \cup \cdots \cup RD_{exit}(\ell_n)$ در بین معادلات جریان داده زمانی که جریان کنترل از بلوک‌های ابتدایی با برچسب‌های $\ell_۱$ تا ℓ_n به بلوک با برچسب ℓ انتقال پیدا می‌کند.

در نهایت مقدار اولیه تحلیل را به‌صورت زیر تعریف کنیم، که بیان کننده این حقیقت است که مکان آخرین مقداردهی

به متغیرهایی که در نقطه شروع برنامه مقداردهی اولیه نشده‌اند معلوم نیست. در صورتی که مجموعه $Var_\star$ را به عنوان مجموعه تمام شناسه‌های متغیرهای مورد استفاده در یک برنامه تعریف کنیم، داریم:

$$RD_{entry}(۱) = \{(x, ?) \mid x \in Var_\star\}.$$

واضح است که داده‌های ارائه شده در جدول ۱.۲ در معادلات جریان داده ارائه شده برای این برنامه صدق می‌کند. با کمی تأمل متوجه می‌شویم که این جواب دقیق‌ترین جواب ممکن برای دستگاه معادلات یاد شده نیز هست. در ادامه راه‌حل نظام‌مندی برای محاسبه کوچکترین جواب برای یک چنین دستگاه معادلات را شرح می‌دهیم.

دستگاه معادلات تنظیم شده برای مثال جاری دوازده مجموعه به‌صورت زیر را بر حسب همدیگر تعریف می‌کند:

$$RD_{entry}(۱), \ldots, RD_{exit}(۶).$$

ما این دوازده‌تایی را با استفاده از نماد $\vec{RD}$ نشان می‌دهیم. به این ترتیب می‌توان دستگاه معادلات را با استفاده از یک تابع F و به‌صورت زیر باز نویسی کرد:

$$\vec{RD} = F(\vec{RD}),$$

که در آن $F(\vec{RD})$ به‌صورت زیر تعریف می‌شود:

$$F(\vec{RD}) = \left(F_{entry}(۱)(\vec{RD}), F_{exit}(۱)(\vec{RD}), \ldots, F_{entry}(۶)(\vec{RD}), F_{exit}(۶)(\vec{RD})\right),$$

به‌طوری که، مثلاً، داریم:

$$F_{entry}(۳)(\ldots, RD_{exit}(۲), \ldots, RD_{exit}(۵), \ldots) = RD_{exit}(۲) \cup RD_{exit}(۵).$$

با تعریف $Lab_\star$ به عنوان مجموعه تمام برچسب‌هایی که در برنامه مثال جاری ظاهر می‌شوند بعلاوه برچسب ویژه $?$، واضح است که می‌توان عملکرد تابع F را به‌صورت زیر تعریف کرد:

$$F : (\mathcal{P}(Var_\star \times Lab_\star))^{۱۲} \to (\mathcal{P}(Var_\star \times Lab_\star))^{۱۲}.$$

همچنین، واضح است که مجموعه $(\mathcal{P}(Var_\star \times Lab_\star))^{۱۲}$ با توجه به ترتیب جزئی زیر یک مجموعه مرتب جزئی است:

$$\vec{RD} \sqsubseteq \vec{RD'} \iff \forall i \in \{۱, \ldots, ۱۲\} \cdot RD_i \subseteq RD'_i,$$

که در آن $\vec{RD} = (RD_۱, \ldots, RD_{۱۲})$ بوده، و نیز $\vec{RD'} = (RD'_۱, \ldots, RD'_{۱۲})$ است. این موضوع مجموعه مورد نظر را به یک مشبک تام تبدیل می‌کند، که کوچک‌ترین عنصر آن $\vec{\emptyset} = (\emptyset, \ldots, \emptyset)$ است. عملگر انجمنی و جابجایی پذیر کوچک‌ترین

کران بالا برای این مشبک نیز به‌صورت زیر تعریف می‌شود:

$$\vec{RD} \sqcup \vec{RD'} = (RD_1 \cup RD'_1, \ldots, RD_{12} \cup RD'_{12}).$$

علاوه بر آن، اثبات یکنوا بودن تابع F نیز ساده است:

$$\vec{RD} \sqsubseteq \vec{RD'} \implies F(\vec{RD}) \sqsubseteq F(\vec{RD'}).$$

به راحتی می‌توان مشاهده کرد که $\vec{\emptyset} \sqsubseteq F(\vec{\emptyset})$. حال دنباله $\left\langle F^n(\vec{\emptyset}) \right\rangle_n$ را در نظر می‌گیریم. از آنجایی که F تابعی یکنوا است، به راحتی با یک استقرای ریاضی می‌توان مشاهده کرد که برای هر n داریم: $F^n(\vec{\emptyset}) \sqsubseteq F^{n+1}(\vec{\emptyset})$. با توجه به عملکرد تابع F متوجه می‌شویم که تمامی عناصر دنباله یاد شده در داخل مجموعه $(\mathcal{P}(Var_\star \times Lab_\star))^{12}$ است، و با توجه به اینکه این مجموعه متناهی است می‌توان گفت که مشبک مورد نظر شرط زنجیر صعودی را برآورده می‌کند. بنابراین، یک n وجود دارد که به ازای آن داریم:

$$F^{n+1}(\vec{\emptyset}) = F^n(\vec{\emptyset}).$$

از آنجایی که $F^{n+1}(\vec{\emptyset}) = F(F^n(\vec{\emptyset}))$ است، می‌توان گفت که $F^n(\vec{\emptyset})$ یک نقطه‌ثابت برای تابع F بوده، و در نتیجه جوابی برای معادله یاد شده است. به راحتی می‌توان مشاهده کرد که این جواب کوچکترین جواب دستگاه معادلات مثال جاری نیز هست. ارائه چند مثال دیگر برای روش‌تر شدن هرچه بیشتر مفاهیم تحلیل جریان داده مفید است، اما در این بخش که هدف مرور مفاهیم اصلی تحلیل ایستا است مجال بیان مثال‌های متعدد نبوده و به همین مثال بسنده می‌کنیم.

در صورتی که تعدادی مثال از تحلیل جریان داده را مشاهده کنیم، متوجه می‌شویم که تمام آن‌ها دارای یک الگوی کلی هستند و به نوعی می‌توان گفت که همه آن‌ها با استفاده از یک چارچوب توصیف شده‌اند. این چارچوب به **چارچوب یکنوا** معروف است که دارای دو جزء زیر است:

- یک مشبک تام L: فضای اطلاعات استخراج شده از متن برنامه‌ها،
- یک مجموعه $\mathcal{F}$ از توابع یکنوا از L به L که شامل تابع همانی بوده و تحت ترکیب توابع بسته است.

تحلیل‌های ارائه شده در این پایان‌نامه نیز نمونه‌هایی از چارچوب یکنوا هستند.

فصل سوم

کارهای مرتبط

در فصل اول، نگاهی بر مهم‌ترین کارهای انجام شده در زمینه کشف خودکار رقابت داده، با تأکید بر روش‌های ایستا، داشتیم. در این فصل، بحثی دقیق‌تر راجع به کارهای مرتبط با پژوهش انجام شده در این پایان‌نامه ارائه می‌شود.

۱.۳ روش‌های مبتنی بر تحلیل جریان داده

در روش‌های مبتنی بر تحلیل جریان داده، همان‌طور که از نامش پیدا است، از تحلیل‌های جریان داده برای استخراج اطلاعات از متن برنامه‌ها استفاده می‌شود. توجه کنید که منظور ما از تحلیل مبتنی بر جریان داده، یک تحلیل مبتنی بر جریان داده غیر بدیهی، تحلیلی تا حد امکان درست و تا حد امکان دقیق، است؛ تحلیل‌های جریان داده در ابزارهای کاربردی مانند RacerX نیز مورد استفاده قرار می‌گیرند، اما تحلیل‌های استفاده شده در این گونه ابزارها نه درست هستند و نه دقیق، و صرفاً برای تخمین به‌کار می‌روند. دقت این‌گونه ابزارها بعد از انجام تحلیل با به‌کارگیری روش‌های ابتکاری و اکتشافی بهبود داده می‌شود. این در حالی است که در روش‌های مبتنی بر تحلیل جریان داده تمام کارها با استفاده از تحلیل‌های جریان داده انجام می‌شود، و کمتر به پس‌پردازش‌های ابتکاری و اکتشافی تکیه می‌شود.

۱.۱.۳ سیستم کشف رقابت داده **IBM**

الگوریتم ارائه شده در [۱۸] از ترکیبی از تحلیل‌های جریان کنترل بین‌ریسه‌ای و نیز تحلیل اشاره‌گری برای کشف رقابت داده در برنامه‌های چندریسه‌ای شی‌گرا استفاده می‌کند. دقت این الگوریتم وابسته به دقت اطلاعات کنترلی و اشاره‌گری استخراج شده از متن برنامه‌ها است. تحلیل جریان کنترل بین‌ریسه‌ای، یک تحلیل مبتنی بر جریان داده بین‌رویه‌ای است که اطلاعات کنترلی را با توجه به مکان‌های ایجاد ریسه و دستورالعمل‌های همگام‌سازی به‌دست می‌آورد.

هدف این الگوریتم، شناسایی آن دسته از دستورالعمل‌ها از یک برنامه است که حداقل یک اجرا از آن برنامه وجود دارد که دستورالعمل‌های مورد نظر اجرا شده و اجرای آن‌ها موجب بروز خطای رقابت داده می‌شود. الگوریتم، به منظور شناسایی چنین دستورالعمل‌هایی به اطلاعات اشاره‌گری با قطعیت may، برای بررسی این موضوع که آیا دو دستورالعمل امکان دسترسی به یک نقطه مشترک حافظه را دارند یا خیر، نیاز دارد. برای بررسی رابطه زمانی بین دو دستورالعمل نیز از

گراف جریان کنترل بین‌ریسه‌ای استفاده می‌شود که در ادامه به توضیح آن می‌پردازیم.

گراف جریان کنترل بین‌ریسه‌ای، اساس کار تحلیل‌های دیگر در سیستم ارائه شده در [۱۸] را تشکیل می‌دهد. سیستم کشف رقابت داده، به کمک این گراف، قادر است از روابط زمانی بین دستورالعمل‌ها اطلاع پیدا کند. از آن‌جایی در گزارش یاد شده اثر دستورالعمل‌های مربوط بع خاتمه ریسه‌ها در نظر گرفته نشده است، یک منبع گزارشات نادرست به کل سیستم اضافه می‌شود. یک گراف جریان کنترل بین‌ریسه‌ای در اصل یک نمایش بین رویه‌ای از یک برنامه چندریسه‌ای است که در آن گره‌ها نشان دهنده دستورالعمل‌ها و کمان‌ها نشان دهنده جریان کنترل است. بدون از دست دادن کلیت مسأله، فرض می‌کنیم که چهار نوع گره داریم: (۱) گره خواندن از یک مکان حافظه، (۲) گره نوشتن در یک مکان حافظه، (۳) گره ورود به مانیتور، و (۴) گره خروج از مانیتور. به گره‌های خواندن یا نوشتن گره‌های دسترسی نیز می‌گویند. به همین ترتیت، فرض می‌کنیم چهار نوع جریان کنترل در گراف جریان کنترل بین‌ریسه‌ای وجود دارد: (۱) درون‌رویه‌ای، (۲) فراخوانی متد، (۳)، برگشت از متد، و (۴) شروع ریسه. همان‌طور که می‌دانیم سه نوع اول از جریان‌ها در یک گراف جریان کنترل بین‌رویه‌ای استاندارد موجود است. جریان‌های شروع ریسه، جریان‌های جدید معرفی شده در [۱۸]، از دستورالعمل‌های ایجاد ریسه به نقطه شروع تمام متدهای فراخوانی شده سازگار تعریف می‌شوند. در زبان برنامه‌سازی جاوا به ازای هر دستورالعمل `e.start();`، و به ازای هر متد `run` فراخوانی شده در آن محل، یک جریان شروع ریسه از دستورالعمل یاد شده به نقطه شروع متد `run` تعریف می‌شود.

با استفاده از گراف جریان کنترل بین‌ریسه‌ای، می‌توان جفت دستوالعل‌ها، از ریسه‌های مختلف، که امکان اجرای همروند را دارند شناسایی کرد. از میان آن‌ها جفت دستورالعمل‌هایی که توسط حداقل یک جفت دستورالعمل مانیتور منطبق بر هم محاصره نشده‌اند به عنوان دستورالعمل‌های مشکوک در به وجود آمدن خطای دقابت داده معرفی می‌شوند. توجه کنید که برای تشخیص منطبق بودن دستورات مانیتور، برای این‌که تحلیل درست باشد، نیاز داریم که از اطلاعات اشاره‌گری با قطعیت must استفاده کنیم.

یک نسخه از این سیستم، که در آن از تحلیل اشاره‌گری مستقل از زمینه استفاده شده است، برای وارسی برنامه‌های زبان برنامه‌سازی جاوا پیاده‌سازی شده است. این پیاده‌سازی برای وارسی چند برنامه کوچک مورد استفاده قرار گرفته است که تنها در یکی از آن‌ها، با وجود به‌کار گیری تحلیل فرار، شش مورد نقص رقابت داده اعلام شده است. لازم به یادآوری است که برنامه یاد شده دارای ۵۴ مورد دستورالعمل دسترسی است.

جمع‌بندی و نتیجه‌گیری: گزارش [۱۸] یک سیستم درست و دقیق برای کشف رقابت داده ارائه می‌دهد. مشکل اساسی تحلیل‌های ارائه شده، مقیاس‌پذیر نبودن آن‌ها است. نیاز به تحلیل اشاره‌گری با دو قطعیت may و must باعث می‌شود که زمان لازم برای وارسی یک برنامه توسط سیستم کشف رقابت داده افزایش یابد.

۲.۱.۳ سیستم کشف رقابت **ETH**

سیستم کشف رقابت داده ETH [۷]، از دو بخش ایستا و پویا تشکیل شده است. بخش پویا، برای تست برنامه‌ها به منظور کشف موارد رقابت شی به‌کار می‌رود، و بخش ایستای سیستم نیز برای تحلیل ایستای برنامه‌ها به منظور تجهیز هدف‌مند برنامه‌ها و کاهش سربار محاسباتی ابزار تست پویا به‌کار می‌رود. در این بخش به بررسی بخش ایستای این سیستم، که مرتبط با پژوهش ما است، می‌پردازیم. توجه کنید که در ادامه منظور ما از سیستم کشف رقابت داده ETH، همان بخش ایستای این سیستم است. این بخش از سیستم در مقاله [۲۳] تشریح شده است.

اساس کار ابزار کشف ایستای رقابت داده، مفهومی به نام گراف استفاده اشیاء، یا همان OUG، است که در بر دارنده اطلاعات دسترسی ریسه‌های مختلف به اشیاء است. گراف استفاده اشیاء، در اصل گسترشی از گراف شکل هیپ است که در آن گره‌ها نشان دهنده شی‌های مجرد و کمان‌ها نشان دهنده رابطه ارجاع بین شی‌ها است. در یک OUG، برای هر گره گراف شکل هیپ، یک مجموعه رخداد و یک ترتیب جزئی بر روی این رخدادها منتسب می‌کند. رخدادهای منتسب شده به یک شی، نشان دهنده خواندن، نوشتن، شروع، و خاتمه ریسه است که توسط ریسه‌های مختلف بر روی شی مورد نظر انجام می‌شوند. این ابزار سریع است، و برای تحلیل برنامه‌ها تنها به چند ثانیه زمان نیاز دارد.

اطلاعات موجود در گراف شکل هیپ مورد استفاده در این پژوهش، حاوی اطلاعات اشاره‌گری و نیز فرار شی‌ها است، و برای کل برنامه معتبر است، یعنی توسط یک تحلیل مستقل از جریان با قطعیت may محاسبه شده است. در این گراف، در صورتی که یک شی یک بار توسط دو ریسه مختلف مورد دسترسی قرار گیرد به عنوان یک شی مشترک بین‌ریسه‌ای در نظر گرفته می‌شود. این در حالی است که دسترسی‌های یاد شده ممکن است توسط دو ریسه که یکی بعد از خاتمه دیگری شروع می‌شود انجام شود. ابزار تحلیل ایستای ETH قابلیت کشف چنین مواردی را دارد. این ابزار با ساخت OUG که تخمینی ایستا از رابطه پیش‌رویدادی بین رخدادهای تولید شده توسط ریسه‌های مختلف است، اقدام به کشف رقابت داده می‌کند. گراف OUG اطلاعات موجود در گراف شکل هیپ را، با افزودن رخدادهای دسترسی به اشیاء مجرد و ترتیب زمانی بین آن‌ها، گسترش می‌دهد. این اطلاعات به اندازه کافی دقیق است که با استفاده از آن‌ها می‌توان، علاوه بر انجام بهینه‌سازی‌های مؤثر، نقص‌های همروندی را کشف کرد.

اطلاعات موجود در OUG اساس بسیاری از کاربردها در کامپایلرهای زبان‌های همروند است. این کاربردها عبارت‌اند از: (۱) گزارش دسترسی‌های مشکوک در شرکت در خطای رقابت داده، (۲) تجهیز هدف‌مند برنامه‌ها به منظور کاهش سربار ابزارهای تست پویا مانند [۹]، (۳) بهینه‌سازی برنامه‌ها با حذف دستورالعمل‌های همگام‌سازی افزونه.

زبان برنامه‌سازی جاوا دارای مدل حافظه ساده‌ای است. شی‌ها در هیپ برنامه اخذ شده و دسترسی به آن‌ها تنها از طریق ارجاع‌هایی امکان‌پذیر است که در زمان ایجاد یک شی محاسبه شده‌اند. این مدل حافظه برای ارائه یک تخمین از هیپ در زمان اجرای برنامه مفید است. این تخمین با استفاده از گراف شکل هیپ برنامه انجام می‌شود. اشیاء مجرد و ریسه‌های مجرد تجریدهای در زمان کامپایل از شی‌ها و ریسه‌های در زمان اجرا است. در ابزار کشف رقابت ETH، گره‌ها گراف شکل هیپ، برای مدل‌سازی اشیاء و ریسه‌های مجرد به‌کار می‌رود. نوع، متدهایی که با ایجاد یک ریسه به عنوان بدنه آن فراخوانی می‌شوند، و نیز چندگانگی ریسه از روی مکان‌های ایجاد ریسه در برنامه استخراج می‌شود.

استفاده از شی‌ها، در گراف OUG، با استفاده از مفهوم رخدادها انجام می‌شود. گره‌های گراف OUG نشان دهنده رخدادها بوده، و کمان‌های آن نشان دهنده تخمینی ایمن از رابطه پیش‌رویدادی است. رخدادها متناظر با دستورالعمل‌های برنامه هستند. انواع رخدادها عبارت‌اند از:

- رخدادهای GET و PUT: برای نشان دادن خواندن و نوشتن در یک فیلد یک شی به‌کار می‌رود.
- رخدادهای $LOAD$، $STORE$، و $ESCAPE$: دو رخداد اول، به ترتیب، برای نشان دادن واکشی ارجاع به یک شی از یک متغیر و ذخیره ارجاع به یک شی در یک متغیر به‌کار می‌رود. رخداد $ESCAPE$ یک نوع $STORE$ است و هنگامی استفاده می‌شود که متغیری که مقصد ذخیره ارجاع است متعلق به شی‌ای باشد که احتمال به اشتراک گذاشته شدن بین چندین ریسه را دارد.
- رخدادهای $TSTART$ و $TJOIN$: این رخدادها برای نشان دادن شروع و خاتمه ریسه‌ها به‌کار می‌رود.

- رخدادهای $ENTRY$ و $EXIT$: این رخدادها برای نشان دادن نقطه ورود و خروج از متدها به‌کار می‌رود. توجه کنید که این رخدادها متناظر با هیچ دستورالعملی در برنامه نیست.

- رخدادهای $CALL$: این رخدادها برای نشان دادن مکان‌های فراخوانی متدها به‌کار می‌روند. توجه کنید که رخدادهای $CALL$ تنها در زمان ساخت OUG مورد استفاده قرار می‌گیرد. اثر متدهای فراخوانی شده در مکان‌های فراخوانی، با جایگذاری اثر متدها با رخدادهای $CALL$، برخط می‌شود. فراخوانی‌های بازگشتی بدون جایگذاری رها می‌شوند.

رخدادها، یا همان گره‌های گراف OUG، دارای چندین خصیصه هستند که به ترتیب عبارت‌اند از ریسه مجرد و مکان برنامه که رخداد مورد نظر را تولید کرده است، ریسه مورد نظر (برای رخدادهای $TSTART$ و $TJOIN$)، شی مقصد (برای رخدادهای $LOAD$، $STORE$، یا $ESCAPE$)، فیلد یا متد، و نیز مجموعه قفل‌هایی که در حین تولید رخداد مورد نظر اخذ شده‌اند (برای رخدادهای GET، PUT، یا $CALL$).

در حالت کلی، سه نوع کمان در OUG برای بیان ترتیب زمانی بین رخدادهای برنامه وجود دارد که عبارت‌اند از:

- کمان‌های انتقال جریان: این کمان‌ها برای نشان دادن جریان کنترل بین‌رویه‌ای به‌کار می‌رود. در صورتی که برنامه‌ای در داخل یک حلقه تکرار و یا تابع بازگشتی به یک شی دسترسی داشته باشد، گراف OUG مربوط به آن شی دارای دور خواهد بود.

- کمان‌های انتقال ارجاع: یک ریسه نمی‌تواند به شی‌ای دسترسی داشته باشد، مادامی که ریسه ایجاد کننده آن ارجاعی از آن را در دسترسی عموم ریسه‌ها قرار داده باشد. این کار می‌تواند از طریق یک شی مشترک بین‌ریسه‌ای انجام دهد. این کمان‌ها از گره‌های متناظر با رخدادهای $ESCAPE$ مربوط به یک شی از یک ریسه به گره‌های متناظر با رخدادهای $LOAD$ مربوط به همان شی در ریسه‌های دیگر تعریف می‌شوند.

- کمان‌های ترتیب ریسه‌ها: از آن‌جایی که هیچ ریسه قبل از ایجاد شدن، و بعد از خاتمه، نمی‌تواند رخدادی تولید کند، ترتیبی بین رخدادهای تولید شده توسط یک ریسه و دستورالعمل‌های ایجاد و خاتمه ریسه‌های در نظر گرفته می‌شود.

دو رخداد در OUG، به عنوان رخدادهای مشکوک در شرکت در خطای رقابت داده شناسایی می‌شوند، هرگاه (۱) ترتیبی بین آن‌ها وجود نداشته باشد، (۲) رخدادها مربوط به ریسه‌های مختلف باشد، (۳) اشتراک مجموعه قفل‌های اخذ شده توسط دو رخداد تهی باشد. سایر رخدادهایی که در این شرایط صدق نمی‌کنند، به عنوان رخدادهای ایمن معرفی می‌شوند. یک تعریف از رقابت داده یک تجرید در زمان اجرا از خطای یاد شده است. از آنجایی که کامپایلر تمامی مسیرهای اجرا را در نظر می‌گیرد، تعدادی از رخدادهای تشخیص داده متناظر با دستورالعمل‌هایی است که ممکن است در هیچ اجرای برنامه اجرا نشوند.

گراف OUG برای یک برنامه با پیمایش گراف شکل هیپ آن و درج رخدادهای تولید شده توسط هر ریسه مجرد در OUG مربوط به شی مجرد دسترسی شده صورت می‌گیرد. این کار با شروع از گره مربوط به شی کلاسِ کلاس اصلی برنامه و بررسی متد `main` برنامه آغاز می‌شود، و همین عمل بررسی به ازای هر گره در گراف شکل هیپ که نشان دهنده یک شی ریسه است، صورت می‌گیرد.

در حالت کلی مراحل تحلیل یک برنامه در ابزار کشف رقابت ETH به‌صورت زیر است:

۱- تعیین ریسه‌های مجرد در برنامه و گراف فراخوانی آن‌ها،

۲- ساخت گراف شکل هیپ برای برنامه،

۳- ساخت گراف OUG، طی یک اجرای نمادین، و

۴- تحلیل گراف OUG، برای شناسایی رخدادهای مشکوک در شرکت در رقابت داده.

هر ریسه مجرد متناظر با یک شی از یک زیرکلاس از کلاس Thread در برنامه است. ریسه مجرد مربوط به ریسه اصلی برنامه نیز متناظر با شی کلاسِ کلاس اصلی برنام است. بعد از شناسایی ریسه‌های مجرد، گراف شکل هیپ با استفاده از یک روش مبتنی بر کلاس هم‌ارزی ساخته می‌شود. به ازای هر ریسه مجرد در برنامه، بدنه متد run متناظر با آن برای محاسبه رخدادهای تولید شده توسط هر ریسه و رابطه زمانی بین آن‌ها تحلیل می‌شود. به این ترتیب، به ازای هر گره در گراف شکل هیپ یک OUG ساخته می‌شود. در نهایت، OUG هر یک از شی‌های مجرد برای یافتن جفت رخدادهای مشکوک در شرکت در رقابت داده‌ها جستجو می‌شود.

جمع‌بندی و نتیجه‌گیری: هدف بخش ایستای ابزار ارائه شده در [۷]، کشف موارد رقابت شی به منظور بهینه‌سازی کارایی بخش پویای آن ابزار است. از این بخش ابزار می‌توان، به‌صورت مستقل، برای کشف رقابت شی استفاده کرد. ابزار یاد شده، از مفهوم رخدادها و ترتیب زمانی بین آن‌ها برای کشف رقابت استفاده می‌کند. گراف استفاده اشیاء فقط برای زبان برنامه‌سازی جاوا طراحی شده و تنها قابلیت در نظر گرفتن همگام‌سازی به روش فقل‌گذاری و شروع و پایان ریسه‌ها را دارد.

سیستم کشف رقابت مورد نظر درست نیست. نادرستی آن بنا چند دلیل است که مهم‌ترین آن‌ها به این شرح است. رخدادهای موجود در گراف استفاده اشیاء منتسب به ریسه‌های مجرد هستند و هر ریسه مجرد با یک شی از یک زیرکلاس Thread، با یک شی مجرد در گراف شکل هیپ، مشخص می‌شود. از طرفی تحلیل مورد استفاده برای ساخت گراف شکل هیپ، یک تحلیل مبتنی بر کلاس هم‌ارزی است. بنابراین، ممکن است تعدادی از گره‌های شکل هیپ، که مربوط به اشیاء ریسه‌ها هستند، در اثر تخمین محافظه‌کارانه تحلیل شکل هیپ از روابط دگرنامی القا شده از سوی برنامه، با هم ادغام شوند. در نتیجه، ممکن است اثر چند نمونه مختلف ریسه در نظر گرفته نشود. با این روش یک ابزار بسیار سریع، اما نادرست، پیاده‌سازی شده است. در نهایت با وجود این‌که سیستم یاد شده پیمانه‌ای و مقیاس‌پذیر است، تحلیل اشاره‌گری مورد استفاده در آن غیر دقیق، و تنها مناسب برای کشف رقابت شی، است.

۳.۱.۳ ابزار **Relay**

ابزار Relay [۲۱]، یک ابزار غیر درست برای کشف ایستای رقابت داده در برنامه‌های زبان C است. این ابزار پیمانه‌ای، و در نتیجه مقیاس‌پذیر، است. هدف از ارائه Relay، ساخت ابزاری مقیاس‌پذیر و تا حد امکان درست برای تحلیل رقابت داده در برنامه‌های زبان برنامه‌سازی C است. در این ابزار منابع نادرستی عبارت‌اند از (۱) خواندن‌ها و نوشتن‌ها در بلوک‌های اسمبلی، (۲) حساب اشاره‌گری، (۳) تحلیل دگرنامی نادرست که توانایی مدل‌سازی جریان‌های داده بین‌رویه‌ای را ندارد، (۴) انجام یک پس‌پردازش مؤثر، اما نادرست، برای صافی کردن گزارش‌های رقابت داده و دسته‌بندی آن‌ها. ارائه این ابزار گامی به سوی ساخت ابزارهای درست و مقیاس‌پذیر برای کشف رقایت داده است.

ابزار Relay، کد برنامه‌ها را برای تضمین این موضوع که آیا برنامه مورد نظر از سیاست قفل‌گذاری پیروی می‌کند یا خیر بررسی می‌کند. یک برنامه در هر دسترسی به یک منبع مشترک باید قفلی را اخذ کند. در این ابزار، از الگوریتم حساس به زمینه و حساس به جریان برای محاسبه مجموعه قفل‌ها و بررسی پیروی برنامه از سیاست قفل‌گذاری استفاده می‌شود.

ایده اصلی ارائه شده در [۲۱]، برای ساخت یک ابزار کشف رقابت مقیاس‌پذیر استفاده از مفهوم مجموعه قفل‌های نسبی است. مجموعه قفل‌های نسبی، به ما اجازه می‌دهد که، مستقل از مکان‌های فراخوانی، خلاصه‌ای از رفتار توابع محاسبه کنیم. برای مثال، خلاصه تابعی مانند f با پارامتر x، مجموعه‌ای از دسترسی‌ها به مکان‌های حافظه قابل دسترسی از اشاره‌گر x به اضافه مجموعه قفل‌هایی که در داخل f در زمان انجام هر دسترسی اخذ شده است. مفهوم مجموعه قفل‌های نسبی، کلید ارائه تحلیلی پیمانه‌ای است. با توجه به این‌که هر یک از توابع برنامه به‌صورت جداگانه تحلیل می‌شوند، می‌توان تحلیل برنامه را در یک سیستم توزیعی و با استفاده از چندین پردازنده به‌صورت موازی انجام داد. با موازی‌سازی تحلیل توابع، هسته چند میلیون خطی سیستم عامل لینوکس در عرض پنج ساعت تحلیل شده است. این در حالی است که وارسی همین کد به‌صورت غیر موازی زمانی حدود هفتاد و دو ساعت نیاز دارد. ابزار برای این کد تعداد ۵۰۲۲ مورد خطای رقابت داده گزارش می‌دهد، که از این بین ۲۵ مورد آن مربوط به خطاهای واقعی است.

اجزای تشکیل دهنده الگوریتم کشف رقابت داده در Relay عبارت‌اند از:

۱- مجموعه قفل‌های نسبی: مجموعه قفل‌های نسبی در نقطه‌ای از یک برنامه، یک دوتایی به‌صورت (L_+, L_-) است که اشتراک دو مجموعه L_+ و L_- همواره تهی است. به زبان ساده، مؤلفه اول هر عنصر نشان دهنده مجموعه قفل‌های اضافه‌ای است که در داخل یک تابع تا نقطه مورد نظر اخذ می‌شوند، و مؤلفه دوم آن نشان دهنده مجموعه قفل‌هایی است که توسط آن تابع تا نقطه مورد آزاد می‌شوند. توجه کنید که برای ارائه یک تحلیل درست نیاز داریم که L_+ و با استفاده از یک تحلیل با قطعیت must محاسبه شده، حال آن‌که نیاز داریم L_- با استفاده از یک تحلیل با قطعیت may محاسبه شود.

۲- دسترسی‌های محافظت شده: یک دسترسی محافظت شده، یک سه‌تایی از دستورالعمل‌ها، مجموعه قفل‌ها، و نوع دسترسی (خواندن یا نوشتن) است که برای هر نقطه از یک برنامه نسبت داده می‌شود. یک دسترسی محافظت شده متناظر با یک دسترسی در بدنه یک تابع است. برای هر تابع در یک برنامه، یک مجموعه از دسترسی‌های محافظت شده محاسبه می‌شود. ابزار Relay، برای هر جفت تابع که هر یک نقطه ورود یک ریسه هستند، مجموعه تمام دسترسی‌های آن‌ها را برای یافتن جفت دسترسی‌های محافظت شده که مؤلفه اول آن‌ها ممکن است به یک مکان مشترک دسترسی داشته باشند، به قسمی که حداقل یکی از آن‌ها از نوع نوشتن باشد، جستجو می‌کند. به ازای هر جفت دسترسی با این شرایط، اگر مجموعه قفل‌های اخذ شده مثبت آن‌ها تهی باشد، یک مورد رقابت داده گزارش می‌شود.

۳- خلاصه توابع: برای محاسبه مجموعه دسترسی‌های محافظت شده برای هر تابع که نقطه ورود یک ریسه است، ابزار Relay گراف فراخوانی را برای برنامه ساخته و با پیمایش پایین به بالای آن مجموعه‌های دسترسی را برای هر یک از توابع ملاقات شده محاسبه می‌کند. برای این منظور، ابزار به ازای هر تابع دو نوع خلاصه محاسبه می‌کند. یکی خلاصه مجموعه قفل‌های نسبی بوده، و دیگری خلاصه دسترسی‌های محافظت شده است. خلاصه متدها به‌صورت پایین به بالا ساخته شده، و در هر مکان فراخوانی خلاصه توابع فراخوانی شده جایگذاری می‌شود.

۴- اجرای نمادین: برای این‌که خلاصه تابع که مستقل از زمینه فراخوانی ساخته می‌شود، در بر گیرنده رفتار کامل تابع باشد، لازم است که بر حسب پارامترهای هر تابع و متغیرهای عمومی بیان شود. به این ترتیب، خلاصه متدها را می‌توان در هر محل فراخوانی نمونه‌سازی کرد.

ابزار Relay وارسی یک برنامه را با پیمایش پایین به بالای گراف فراخوانی آن انجام می‌دهد. در حین پیمایش، از بین تمام توابعی که ملاقات می‌شود آن تابعی که تمام توابع فراخوانی شده توسط آن پیش‌تر تحلیل شده است، تحلیل می‌شود. برای تحلیل هر برنامه، ابزار به ترتیب سه نوع تحلیل انجام می‌دهد: (۱) اجرای نمادین، (۲) تحلیل مجموعه قفل‌های نسبی، و (۳) تحلیل دسترسی‌های محافظت شده. اجرای نمادین برای این انجام می‌شود که آدرس‌های حافظه‌ای که یک تابع دستکاری می‌کند، بر حسب پارامترهای آن و متغیرهای عمومی بیان شود. این اطلاعات نمادین برای دو تحلیل بعد لازم است. تحلیل مجموعه قفل‌های نسبی یک تحلیل مبتنی بر جریان داده است که مجموعه قفل‌های اخذ شده برای هر نقطه از برنامه محاسبه می‌کند. در هر مکان فراخوانی، تحلیل از خلاصه توابع فراخوانی شده برای تعیین مجموعه قفل‌ها بعد از دستورالعمل فراخوانی استفاده می‌کند. بعد از محاسبه نقطه ثابت، مجموعه قفل‌های اخذ شده منتسب به برچسب نهایی تابع به عنوان خلاصه مجموعه قفل‌های نسبی تابع استفاده می‌شود. بعد از انجام تحلیل مجموعه قفل‌های نسبی، Relay تحلیل دسترسی‌های محافظت شده را بر روی همان تابع انجام می‌دهد. تحلیل دسترسی‌های محافظت شده نیز یک تحلیل مبتنی بر جریان داده است که مجموعه دسترسی‌های انجام شده تا هر نقطه از بدنه یک تابع را به نقاط یاد شده نسبت می‌دهد.

با توجه به این‌که Relay یک ابزار کشف رقابت مبتنی بر مجموعه قفل‌ها است، دقت کافی در تشخیص بسیاری از اصطلاحات همگام‌سازی را ندارد. بنابراین، تعداد زیادی از موارد گزارش شده توسط این ابزار گزارشات نادرست است. برای مثال، از میان ۵۰۲۲ مورد خطای گزارش شده تنها ۹۰ مورد آن مربوط به رقابت‌های داده واقعی است. مواردی که موجب می‌شوند که ابزار گزارش نادرست تولید کند، به شرح زیر است.

۱- مقداردهی اولیه: یک اصطلاح معمول برنامه‌سازی این است که ساختمان داده‌ای که در ریسه‌ای ایجاد می‌شود، قبل از اینکه در معرض دسترسی ریسه‌های دیگر قرار گیرد، بدون انجام هیچ‌گونه همگام‌سازی بین‌ریسه‌ای مقداردهی اولیه می‌شود. از آنجایی که این عملیات بدون اخذ قفل انجام می‌شوند، با وجود این‌که ایمن و بدون رقابت هستند، به عنوان موارد نقص رقابت داده گزارش می‌شوند.

۲- دگرنامی‌های غیرممکن: ابزار Relay از یک تحلیل دگرنامی محافظه‌کار و غیر دقیق استفاده می‌کند، به‌طوری که تمامی فیلدهای یک ساختمان داده، تمامی عناصر یک آرایه، و حتی تمامی مکان‌های حافظه از یک نوع خاص که یک حساب اشاره‌گری از آن نوع در برنامه انجام گرفته است به عنوان دگرنام شناسایی می‌شود. همچنین، جریان‌های داده بین‌رویه‌ای نادیده گرفته می‌شود. بنابراین، تحلیل دگرنامی ارائه شده، علاوه بر این‌که یک منبع نادرستی است، موجب می‌شود که تعداد زیادی خطای رقابت داده بی‌مورد توسط ابزار گزارش شود.

۳- اشتراک کاذب: بسیاری از دسترسی‌ها به این صورت است که شی دسترسی شده با وجود این‌که بین‌ریسه‌ها مشترک است، در زمان استفاده صرفاً توسط یک ریسه مورد دسترسی قرار می‌گیرد. برای مثال، اگر ریسه‌ای به این صورت به اشیاء موجود در یک لیست مشترک دسترسی پیدا می‌کند که قبل از استفاده از هر شی‌ای آن را از لیست یاد شده حذف می‌کند و بعد از استفاده آن را دوباره در لیست قرار می‌دهد. به این ترتیب، تمام دسترسی انجام شده بر شی‌های لیست مورد نظر به‌صورت انحصاری صورت می‌گیرد.

۴- سمافورها: سمافورها ساختارهایی هستند که یک قفل بعد از k بار اعمال دستورالعمل اخذ، اخذ شده و قفل اخذ شده تنها بعد از k بار اعمال دستورالعمل آزادسازی قفل آزاد می‌شود. از آن‌جایی که Relay به‌صورت محافظه‌کارانه با مشاهده اولین دستورالعمل آزادسازی، قفل مورد نظر را آزاد شده فرض می‌کند تعداد زیادی از عملیات انجام شده

بعد از اولین دستورالعمل آزادسازی قفل به عنوان موارد رقابت داده گزارش می‌شوند.

۵- ریسه‌های غیرهمروند: بسیاری از دسترسی‌ها توسط گردایه‌ای از ریسه‌ها انجام می‌شود که در هر لحظه تنها یکی از آن‌ها فعال است. بنابراین، دسترسی‌های صورت گرفته در این ریسه‌ها نیازی به همگام‌سازی ندارند، که این باعث می‌شود Relay تعداد زیادی گزارش بی‌مورد رقابت داده تولید کند.

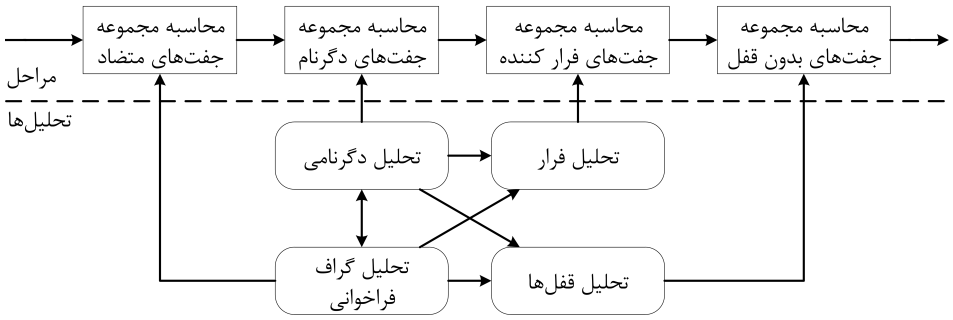
برای جلوگیری از نمایش تعداد زیادی گزارش رقابت داده که بسیاری از آن‌ها بی‌مورد هستند، از روش‌های ابتکاری برای حذف زیرمجموعه‌ای از آن‌ها استفاده می‌کند. این صافی پیام‌های خطا موجب نادرست شدن ابزار می‌شود، اما تجربه نشان داده است که یک چنین صافی می‌تواند کمک زیادی در حذف گزارشات نادرست و ارائه موارد واقعی کند. برای هر یک از منابع گزارشات نادرست که پیش‌تر فهرست کردیم، کارهای زیر برای حذف خطاهای به‌وجود آمده از آن‌ها انجام می‌شود:

- مقداردهی اولیه: برای حذف گزارشات ناشی از دسترسی‌های بدون همگام‌سازی در حین مقداردهی اولیه اشیاء در ریسه‌های ایجاد کننده آن‌ها، تمام گزارشات خطا که مربوط شی‌هایی می‌شوند که این شی‌ها در یکی از ریسه‌های تولید کننده دسترسی‌های متضاد مورد نظر ایجاد شده‌اند.
- سمافورها: از آنجایی که سمافورها بیشتر در ریسه boot-up ایجاد شده در هسته سیستم‌عامل لینوکس استفاده شده‌اند، تمام خطاهایی که در آن یکی از دسترسی‌ها مربوط به این ریسه می‌شوند را در نظر نگیریم.

صافی‌های دیگری نیز برای حذف سایر خطاها طراحی شده است که در مقاله [۲۱] به جزئیات آن‌ها اشاره‌ای نشده است.

جمع‌بندی و نتیجه‌گیری: ابزار Relay یک سیستم کشف رقابت داده برای برنامه‌های نوشته شده به زبان C است. این سیستم با هدف ساخت یک ابزار کشف ایستای رقابت داده درست و مقیاس‌پذیر طراحی شده است. این ابزار به دلیل طبیعت زبان برنامه‌سازی C و نیز عدم استفاده از یک تحلیل دگرنامی درست و دقیق، به‌طور کامل، به هدف درستی دست پیدا نکرده است و حتی هیچ اثباتی برای زیرمجموعه‌ای محدودتر از زبان هدف، با فرض درست بودن تحلیل دگرنامی، انجام نشده است. دقت ابزار Relay به دلیل در نظر نگرفتن روابط زمانی بین دستورالعمل‌ها بسیار پایین است. بنابراین، طراح برای کاهش تعداد گزارشات نادرست مجبور شده است که از یک صافی گزارشات خطا استفاده کند. همان‌طور که مشاهده شد، صافی یاد شده خود منبع نادرستی برای ابزار است، چرا که ممکن است بسیاری از گزارشات واقعی رقابت داده را به اشتباه حذف کند. از طرفی ابزار یاد شده برای وارسی تنها یک برنامه، هسته سیستم عامل لینوکس، مورد استفاده قرار گرفته است، و همان‌طور که مشاهده کردیم بخش‌هایی از ابزار به‌طور انحصاری برای وارسی این برنامه خاص ساخته شده است. واضح است که این طراحی برای سایر برنامه‌ها نتیجه مطلوبی نخواهد داشت. در نهایت، تحلیل‌های دگرنامی با دو قطعیت may و must زمان اجرای ابزار را به میزان قابل توجهی افزایش می‌دهند، زیرا تحلیل‌های must اصولاً حساس به مسیر هستند (مثلاً برای بررسی مسیرهای مختلف شرط‌ها) و نیاز به اطلاعات اضافی مانند اطلاعات فرار و چندگانگی اشیاء دارند.

مشکل نادرست بودن تحلیل دگرنامی، بدون در نظر گرفتن حساب اشاره‌گرها در زبان C، با مدل‌سازی صحیح جریان‌های داده بین‌رویه‌ای حل می‌شود. همچنین، مشکل گزارشات نادرست، ناشی از دقت پایین روش مجموعه قفل‌ها برای کشف رقابت داده، با در نظر گرفتن روابط زمانی، و نیز استفاده از رخدادها برای مدل‌سازی اثر دستکاری سمافورها، قابل حل است. به این ترتیب نیازی، هم به استفاده از صافی گزارشات نیست.



شکل ۱.۳: معماری ابزار Chord

۴.۱.۳ ابزار **Chord**

الگوریتم ارائه شده در [۱۸]، چند سال بعد اساس کار یکی از سریع‌ترین و دقیق‌ترین ابزارهای کشف رقابت برای زبان جاوا، یعنی Chord [۱۹، ۲]، قرار گرفت. تفاوت اساسی الگوریتم مورد استفاده در Chord با الگوریتم ارائه شده در [۱۸] این است که Chord از یک تحلیل دگرنامی حساس به متن استفاده می‌کند. اصول کار این ابزار به این صورت است که جفت دستورالعمل‌های مشکوک به شرکت در رقابت در یک برنامه با گذر از مراحلی پالایش می‌شوند، به‌طوری که در هر مرحله دستورالعمل‌هایی که ایمن بودن اجرای همروند آن‌ها اثبات می‌شود از مجموعه مورد نظر حذف می‌گردند.

تحلیل دگرنامی مورد استفاده در Chord، از مفهوم حساسیت k-سطحی برای تفکیک زمینه‌های مختلف فراخوانی استفاده می‌کند. حساسیت k-سطحی به این صورت است که هر مکان فراخوانی با دنباله‌ای به‌طول حداکثر k از شی‌های مجرد که پارامتر صفرام (متغیر ویژه `this`) متد(های) فراخوانی شده در آن مکان می‌تواند به آن اشاره کند شناسایی می‌شود. حساس به زمینه بودن تحلیل دگرنامی کلید دقت ابزار Chord است. انتخاب این شیوه از حساسیت به متن بعد از چندین آزمایش انتخاب شده است. نتایج این آزمایش‌ها نشان می‌دهد که تحلیل دگرنامی با حساسیت k-سطحی دقیق‌تر از سایر انواع تحلیل‌های دگرنامی حساس به زمینه است. اما استفاده از یک تحلیل دگرنامی حساس به زمینه مشکلات مقیاس‌پذیری فراوانی را به دنبال دارد. برای بهبود مقیاس‌پذیری ابزار از نمودارهای تصمیم دودویی استفاده شده است.

الگوریتم کشف رقابت مورد استفاده در [۱۹] حساس به زمینه، اما مستقل از جریان، است. این استقلال از جریان در حالت کلی موجب کاهش دقت ابزار کشف رقابت پیاده‌سازی می‌شود. بر خلاف ساختارهایی مانند `synchronized`، تشخیص اثر ساختارهای همگام‌سازی `start` و `join` با استفاده از یک تحلیل مستقل از جریان ممکن نیست.

سیستم کشف رقابت در Chord به‌صورت نشان داده شده در شکل ۱.۳ است. همان‌طور که مشاهده می‌شود ابزار از دنباله‌ای از فازهای پردازشی تشکیل شده است. که این فازها برای انجام پردازش به تحلیل‌های گراف جریان کنترل، دگرنامی، فرار، و قفل استفاده می‌کنند. در ادامه به معرفی هر یک از این فازهای پردازشی می‌پردازیم.

در مرحله محاسبه جفت‌های متضاد، با استفاده از اطلاعات کنترلی حاصل از تحلیل گراف جریان کنترل و به کمک یک تحلیل جریان داده بین‌رویه‌ای جفت دستورالعمل‌های متضاد شناسایی می‌شود. لازم به یادآوری است که در این‌جا منظور ما جفت دستورالعمل‌های متضاد، جفت دستورالعمل‌های نوشتن و دسترسی (خواندن یا نوشتن) بر روی یک فیلد مشترک است. نتیجه این مرحله، اولین تخمین از جفت دستورالعمل‌های مشکوک در به وجود آمدن رقابت داده است. خروجی مرحله محاسبه جفت‌های متضاد، وارد مرحله محاسبه جفت‌های دگرنام می‌شود. در این مرحله، به کمک اطلاعات دگرنامی حاصل

از تحلیل دگرنامی، جفت دستورالعمل‌های متضادی که ممکن نیست به یک شی دسترسی داشته باشند از مجموعه جفت دستورالعمل‌های مشکوک به شرکت در رقابت داده حذف می‌شوند. در مرحله بعد، آن دسته از جفت دستورالعمل‌هایی حداقل یکی از مؤلفه‌های آن‌ها تنها به شی‌هایی دسترسی دارند که در تمام مدت حیات خود در حوزه دید یک ریسه بخصوص قرار دارند (فرار نمی‌کنند)، از مجموعه جفت دستورالعمل‌های مشکوک به شرکت در رقابت داده حذف می‌شوند. همان‌طور که در شکل مشاهده می‌شود اطلاعات فرار از طریق یک تحلیل فرار تأمین می‌شود.

تحلیل قفل، یک تحلیل حساس به زمینه و حساس به جریان است که، برای محاسبه مجموعه قفل‌های اخذ شده در هر نقطه از برنامه به‌کار می‌رود. با استفاده از اطلاعات به‌دست آمده از این تحلیل، می‌توان تخمینی ایمن از مجموعه قفل‌های اخذ شده در حین اجرای یک دستورالعمل به‌خصوص به‌دست آورد. توجه کنید که درستی کل سیستم به نوع تحلیل دگرنامی استفاده شده در این تحلیل دارد و برای اینکه سیستم کشف رقابت داده درست باشد باید از یک تحلیل دگرنامی با قطعیت must استفاده شود. نسخه اول Chord [۱۹] از یک تحلیل دگرنامی با قطعیت may استفاده می‌کند، و در نتیجه درست نیست، حال آن‌که نسخه دوم این ابزار [۲۰] با به‌کارگیری یک تحلیل دگرنامی خاص منظوره مشکل نادرستی ابزار را حل می‌کند. در آخرین مرحله پردازش‌ها، آن دسته از جفت دستورالعمل‌هایی که مجموعه قفل‌های اخذ شده برای آن‌ها غیر تهی است از مجموعه جفت دستورالعمل‌های مشکوک به شرکت در رقابت داده حذف می‌شوند. باقی جفت دستورالعمل‌ها، بعد از یک پس پردازش، به عنوان جفت دستورالعمل‌های مشکوک به شرکت در رقابت داده گزارش می‌شوند

دقت این ابزار بعداً با ارائه یک تحلیل دگرنامی خاص منظوره و دقیق‌تر تقویت شد [۲۰]. نسخه نهایی ابزار Chord درست بوده و دارای گزارشات نادرست کمی است، همچنین تلاش بسیاری برای افزایش مقیاس‌پذیری آن صورت گرفته است [۲]. با این حال این ابزار نیاز به وجود کل برنامه دارد و این می‌تواند در بسیاری از موارد محدود کننده باشد، و اصولاً مقیاس‌پذیر نیست [۴].

جمع‌بندی و نتیجه‌گیری: دقت ابزار Chord بالا، سرعت آن قابل قبول، و نسخه نهایی آن نیز درست است. تنها مشکل جدی این ابزار مقیاس‌پذیر نبودن آن، به دلیل تحلیل کل برنامه به‌صورت یک‌جا، است [۲، ۴]. نتایج آزمایش‌ها، در [۲۴]، نشان می‌دهد که این ابزار قادر به تحلیل برنامه‌های بزرگتر از پنج هزار خط را ندارد. ابزار Chord، فقط برای زبان جاوا طراحی شده و تنها قابلیت تشخیص اصطلاحات همگام‌سازی بین‌ریسه‌ای قفل‌گذاری و شروع و پایان ریسه‌ها را دارد.

تحلیل برنامه‌های باز در Chord نیز به این صورت است که یک برنامه مصنوعی برای فراخوانی متدهای برنامه باز مورد نظر، در شرایط مختلف، ساخته می‌شود. اندازه این برنامه مصنوعی، که harness نام دارد، می‌تواند به‌صورت نمایی بزرگ شود [۱۹]. با وجود این‌که امکان تقلیل اندازه برنامه harness، با استفاده از روش‌های وارسی مدل وجود دارد، این کار پرهزینه ضروری به نظر نمی‌رسد [۷].

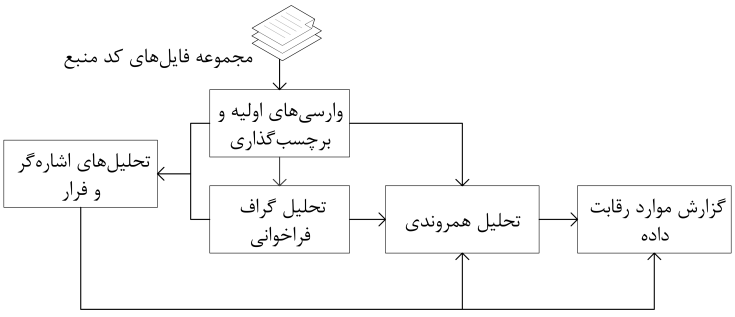
فصل چهارم

روش ارائه شده برای کشف رقابت داده

در این فصل ابتدا با ارائه معماری یک سیستم کشف رقابت داده، در مورد کشف ایستای رقابت داده، به‌صورت کلی، بحث می‌کنیم. سپس، زبان مورد استفاده برای معرفی الگوریتم‌های پیشنهادی در این پایان‌نامه معرفی می‌شود. بعد از آن یک تحلیل همروندی نوین ارائه می‌شود، و در نهایت در مورد کشف رقابت داده با استفاده از این تحلیل بحث می‌کنیم. پیوست دوم پایان‌نامه حاوی اثبات درستی تحلیل بین‌رویه‌ای است.

۱.۴ معماری سیستم

شکل ۱.۴ معماری یک سیستم کشف ایستای رقابت داده را نمایش می‌دهد. همان‌طور که مشاهده می‌شود این معماری به روش صافی و خط لوله توصیف شده است. هر صافی دارای یک یا چندین ورودی و یک خروجی است، که پردازشی بر روی ورودی‌های دریافت شده انجام داده و نتیجه را به خروجی منتقل می‌کند. هر صافی تا زمانی که تمامی ورودی‌هایش آماده نباشد پردازش خود را شروع نمی‌کند.



شکل ۱.۴: معماری یک سیستم کشف ایستای رقابت داده، که در آن هر مستطیل نشان دهنده یک فاز پردازشی از سیستم است

ورودی کل سیستم گردایه‌ای از فایل‌های کد منبع است. در این مرحله برای ما اهمیتی ندارد که این فایل‌ها حاوی چه چیز و به چه زبانی هستند. تنها فرضی که در مورد این زبان می‌کنیم، این است که نوع‌دار و نوع‌ایمن است؛ این یعنی وارسی

نوع زبان قادر است تا تمامی نقص‌های برنامه را که سیستم‌نوع زبان برای مقابله با آن‌ها در نظر گرفته شده است، کشف کند. صافی وارسی اولیه و برچسب‌گذاری شامل تحلیل نحوی و تحلیل معنایی پایه‌ای است. در این پایان‌نامه فرض ما بر این است که تحلیل معنایی پایه‌ای زبان مورد بررسی به اندازه کافی مؤثر باشد به‌طوری که خطاهایی مانند دسترسی به حافظه با استفاده از اشاره‌گرهای تهی و نیز شروع مجدد یک ریسه توسط آن برطرف شود. در این صافی، همچنین، کد منبع ورودی به شیوه‌ای برچسب‌گذاری می‌شود که هر عبارت، دستورالعمل، و اعلان موجود در برنامه ورودی دارای یک برچسب منحصر به‌فرد باشد و هر کدام از آن‌ها با یک و تنها یک برچسب قابل شناسایی باشند. خروجی این صافی یک برنامه برچسب‌گذاری شده خالی از خطاهای یاد شده است.

صافی تحلیل گراف فراخوانی، همان‌طور که از نامش پیدا است، یک تحلیل گراف فراخوانی یا همان تحلیل جریان کنترل برای تعیین ایستای این‌که هر رویه (یا تابع) در برنامه چه رویه‌هایی (یا توابع) را ممکن است فراخوانی کند، و نیز هر مکان فراخوانی در بدنه یک رویه (یا تابع) چه رویه‌ها (یا توابع) را ممکن است فراخوانی کند. توجه کنید که حل این مسأله در زبان‌های شئ‌گرا و زبان‌هایی که با توابع به‌عنوان شهروندان درجه اول زبان برخورد می‌شود، دشوار بوده و تنها با یک بررسی شناسه‌های مورد استفاده در دستورالعمل‌های فراخوانی حل نمی‌شود. خروجی این صافی یک گراف فراخوانی (گرافی جهت‌دار) است که گره‌های آن رویه‌های (یا توابع) برنامه ورودی که تشکیل شده از مجموعه‌ای از مکان‌های فراخوانی در بدنه رویه (یا تابع) مورد نظر است. کمان‌های گراف فراخوانی رابطه ممکن‌است‌فراخوانی‌کند را بین مکان‌های فراخوانی و رویه‌ها (یا توابع) بیان می‌کند.

صافی تحلیل اشاره‌گر و فرار، همان‌طور که از نامش پیدا است، حاوی یک تحلیل اشاره‌گر به همراه یک تحلیل فرار است. تحلیل اشاره‌گر برای تعیین ایستای این که هر اشاره‌گر (بعد از ارزیابی) به چه مکان‌هایی ممکن است اشاره کند، به‌کار می‌رود. تحلیل فرار نیز برای تعیین ایستای حوزه قابلیت دیده شدن اطلاعات موجود هر متغیر به‌کار می‌رود. در یک زبان شئ‌گرا تحلیل اشاره‌گر، به‌صورت ایستا، تعیین می‌کند که هر عبارت ممکن است به کدام شئ‌هایی که در برنامه ایجاد شده‌اند ارزیابی شود. در یک زبان شئ‌گرای همروند تحلیل فرار، به‌صورت ایستا، برای تعیین قابلیت مشاهده شدن (توسط ریسه‌های ایجاد شده در برنامه) شئ‌های ایجاد شده در هر ریسه می‌تواند به‌کار رود. یک سیستم کشف ایستای رقابت داده، از تحلیل اشاره‌گر برای تعیین عبارت‌های دگرنام، و از تحلیل فرار برای نشان تعیین این موضوع که شئ‌های مورد استفاده توسط یک ریسه قابلیت مشاهده و استفاده شدن توسط ریسه‌های دیگر را دارد یا خیر استفاده می‌کند. خروجی این صافی برای یک زبان شئ‌گرای همروند دو تابع است: یکی برای تعیین مجموعه مکان‌های اخذ شئ که یک عبارت ممکن است به اشیاء ایجاد شده در آن محل‌ها اشاره کند، و دیگری برای تعیین اینکه آیا یک شئ توسط بین از یک ریسه قابل مشاهده شدن است یا خیر به‌کار می‌رود. در ادبیات تحلیل اشاره‌گر و نیز در این پایان‌نامه تابع اول را با $mayPointsTo$ نشان می‌دهیم. تابع دوم را در ادبیات تحلیل فرار با استفاده از نماد $captured$ نشان می‌دهند. از آن‌جایی که تحلیل فرار برای کشف رقابت داده ضروری نیست و تنها برای افزایش دقت یک ابزار به‌کار می‌رود، در این پایان‌نامه هیچ تحلیل فراری ارائه نشده است. اما قرار دادن آن در کنار یک تحلیل اشاره‌گر کار دشواری نیست [۳۶].

صافی تحلیل همروندی با دریافت یک برنامه برچسب‌گذاری شده، گراف فراخوانی آن برنامه، و اطلاعات اشاره‌گری اقدام به تحلیل همروندی می‌کند. تحلیل همروندی برای تعیین ایستای جفت دستورالعمل‌ها یا عبارات یک برنامه که ممکن است به‌صورت همروند اجرا شوند به‌کار می‌رود. یک سیستم کشف رقابت داده با استفاده از اطلاعات همروندی و دگرنامی آن جفت دستورالعمل‌ها یا عبارت را که ممکن است به یک نقطه حافظه دسترسی داشته باشند را شناسایی می‌کند. خروجی این صافی یک رابطه، معروف به رابطه ممکن‌است‌همزمان‌رخ‌دهند یا MHP، است. این رابطه حاوی جفت دستورالعمل‌ها یا

عبارات است که ممکن است به‌صورت همزمان اجرا و یا ارزیابی شوند.

صافی گزارش موارد رقابت داده با دریافت رابطه MHP و اطلاعات اشاره‌گری (و فرار در صورت وجود) اقدام به گزارش موارد رقابت داده، در صورت مشاهده زوج مرتبی در MHP که به یک مکان مشترک حافظه دسترسی دارند، می‌کند. در صورتی که ابزار برای هر برنامه که حداقل یک مورد نقص رقابت داده دارد، حداقل یک مورد وجود این نقص را گزارش دهد، گوییم ابزار درست است. میزان گزارشاتی که مربوط به موارد رقابت داده واقعی (آن‌هایی که در حداقل یک اجرای برنامه بروز می‌کنند) هستند نیز دقت ابزار را نشان می‌دهد. خروجی این صافی فهرستی از گزارشات خطا است. در صورتی که چنین سیستم کشف رقابت داده‌ای در یک کامپایلر گنجانده شود، ممکن است طراح تصمیم بگیرد که در صورت وجود حداقل یک مورد گزارش خطا در فهرست یاد شده از ترجمه برنامه ورودی خودداری کند.

در این پایان‌نامه هدف ارائه یک سیستم پیمانه‌ای کشف رقابت داده است. پیمانه‌ای بودن یک تحلیل به این معنا است که هر بخش یک برنامه فقط برای یک بار و آن هم در انزوا و بدون نیاز به کل برنامه تحلیل می‌شود، و تحلیل کل برنامه با ترکیب جواب به‌دست آمده برای هر یک از بخش‌های برنامه میسر می‌شود. از آنجایی که بالاترین میزان پیچیدگی در محاسبات در محاسبه جواب برای یک برنامه با حل معادلات استخراج شده برای آن است، روش پیشنهاد شده از این جهت که مسأله حل معادله را به زیر بخش‌های برنامه محدود می‌کند مقیاس‌پذیر است. این یعنی برنامه ورودی با هر اندازه‌ای که برای ابزار فراهم شود، به زیربخش‌هایی (کوچکتر) تقسیم شده و پاسخ تحلیل کل برنامه را با ترکیب این زیربخش‌ها به‌دست می‌آورد. در فصل مربوط به پیاده‌سازی و آزمایش مشاهده خواهیم کرد که می‌توان یک چنین معماری را برای ساخت یک ابزار پیمانه‌ای کشف رقابت داده مورد استفاده قرار داد. همان‌طور که پیش‌تر نیر اشاره کردیم، در این پایان‌نامه تحلیل فرار ارائه نشده است. در عوض یک تحلیل اشاره‌گر پیمانه‌ای دقیق مبتنی بر تحلیل معرفی شده در [۳۶] ارائه شده است، که به راحتی می‌توان به موازات آن یک تحلیل فرار انجام داد. الگوریتم‌های ارائه شده در این پایان‌نامه قابل استفاده برای هر زبان شئ‌گرا و همروند است، اما برای نمایش این الگوریتم‌ها از یک زیرمجموعه از زبان برنامه‌سازی جاوا [۱] استفاده شده است. با توجه به این‌که در زبان برنامه‌سازی جاوا یک برنامه تشکیل شده از گردایه‌ای از کلاس‌ها است، و از آنجایی که هر کلاس حاوی تعدادی متد است، در این پایان‌نامه زیربخش‌هایی که برنامه را به آن‌ها تقسیم می‌کنیم متدها هستند. فرض ما بر این است که برنامه‌نویس برنامه خود را طوری نوشته است که تعداد خطوط برنامه مورد نظر به‌صورت متعادل بین متدهای مختلف برنامه توزیع شده است. در این حالت است که تفاوت سیستم پیمانه‌ای ارائه شده در این پایان‌نامه و ابزارهای غیرپیمانه‌ای موجود پررنگ می‌شود. با این حساب بدترین حالت (مقیاس‌پذیری) برای الگوریتم‌های ما حالتی است که برنامه‌های ورودی تشکیل شده از تنها یک متد هستند. در این حالت هر ابزار پیاده‌سازی شده توسط این الگوریتم‌ها (از لحاظ مقیاس‌پذیری) مانند سایر ابزارهای موجود عمل خواهد کرد.

در این پایان‌نامه فرضیاتی در مورد چند مجموعه و تابع صورت گرفته است، ترجیح می‌دهیم که این فرضیات را در همین ابتدا به‌صورت صریح بیان کنیم. هر چند که در ادامه این فصل و نیز در فصول دیگر تعریف دقیق هر یک از این مجموعه‌ها و توابع را مشاهده خواهیم کرد. مجموعه $Identifiers$ را به عنوان مجموعه‌ای شمارا از تمام شناسه‌های مجاز برای کلاس‌ها، فیلدها، متدها، و متغیرها تعریف می‌کنیم. بدیهی است که کلمات کلیدی (رزرو شده) زبان جاوای مورد بررسی نباید در این مجموعه قرار بگیرند. مجموعه $Labels$ را به عنوان مجموعه‌ای شمارا از برچسب‌های مورد استفاده برای برچسب‌گذاری برنامه ورودی فرض می‌کنیم (در ادامه این فصل، در مورد این مجموعه بیش‌تر بحث می‌کنیم). مجموعه N را به عنوان مجموعه اشیاء مجردی (مکان‌های ایجاد شئ) که یک عبارت زبان جاوای مورد بررسی ممکن است اشاره کند، تعریف می‌کنیم (در فصل مربوط به تحلیل اشاره‌گر مورد این مجموعه بیش‌تر بحث می‌کنیم). با توجه به این مجموعه‌ها دو مجموعه دیگر را تعریف

می‌کنیم: (۱) مجموعه $CallSites$ که زیرمجموعه محضی از $Labels$ است، برای نشان دادن مجموعه مکان‌ها فراخوانی متد و ایجاد ریسه (در بدنه متدها) به‌کار می‌رود؛ (۲) مجموعه $Methods$ که زیرمجموعه محضی از حاصل ضرب دکارتی مجموعه $Identifiers$ با خودش است، برای نشان دادن مجموعه اسامی کامل متدها در یک برنامه (با استفاده از جفت شناسه کلاس، و شناسه متد اعلان یا به ارث برده شده توسط آن کلاس) به‌کار می‌رود. در نهایت مجموعه $Expressions$ را به عنوان مجموعه تمام عبارت‌های زبان جاوای مورد بررسی معرفی می‌کنیم. در این فصل (به جز تنها یک جا که توضیحات کافی در همان‌جا ارائه شده) وارد جزئیات تحلیل‌های گراف فراخوانی و اشاره‌گرد نخواهیم شد، و فرض می‌کنیم که نتیجه این دو تحلیل را در دست داریم. با استفاده از مجموعه‌های معرفی شده، عملکردی برای دو تابع زیر تعریف می‌کنیم که فرض ما بر این است که آن‌ها را که به عنوان نتیجه تحلیل جریان گراف فراخوانی در اختیار داریم:

$$callees : CallSites \to \mathcal{P}(Methods),$$
$$mayCall : Methods \to \mathcal{P}(Methods).$$

تابع $callees$ با دریافت یک مکان فراخوانی متد یا مکان ایجاد ریسه، مجموعه اسامی کامل تمام متدهایی که در آن نقطه ممکن است فراخوانی شوند را برمی‌گرداند. تابع $mayCall$ نیز با دریافت نام کامل یک متد، مجموعه اسامی کامل تمام متدهایی که متد مورد نظر ممکن است فراخوانی کند را برمی‌گرداند. با توجه به توصیفی که از گراف فراخوانی داشتیم، و نیز همان‌طور که در ادامه در فصل مربوط به تحلیل جریان کنترل و اشاره‌گر خواهیم دید، به راحتی می‌توان این دو تابع را برحسب گراف فراخوانی یک برنامه تعریف کرد.

همان‌طور که پیش‌تر نیز گفتیم، تابع $mayPointsTo$ را برای نشان دادن مجموعه شئ‌هایی که یک عبارت ممکن است ارزیابی شود تعریف می‌کنیم. با استفاده از مجموعه‌های معرفی شده در پاراگراف قبل می‌توان عملکرد این تابع را به‌صورت دقیق‌تر برحسب مجموعه عبارات زبان جاوای مورد بررسی و مجموعه اشیاء مجرد، به‌صورت زیر تعریف کرد:

$$mayPointsTo : Expressions \to \mathcal{P}(N).$$

آخرین نکته‌ای که نیاز به تشریح دارد این است که در زبان جاوای مورد بررسی می‌توان بدون از دست دادن کلیت مسأله فرض کرد که تمام عبارت‌های به‌کار رفته در یک برنامه در داخل یک متد قرار دارد. این فرض کمک می‌کند که تحلیل‌ها را به‌صورت نظام‌مند (یک شکل و دارای قاعده یکسان برای مقابله با تمام حالت‌ها) ارائه دهیم. نماد Val نام دیگری برای مجموعه $\mathcal{P}(N)$ است.

۲.۴ زبان جاوای همروند

در این بخش نحو مجرد و معناشناخت زیرمجموعه‌ای از زبان جاوا ارائه شده است (برای مشاهده معناشناخت ایستای زبان به پیوست اول رجوع کنید). این زبان در عین حال که ساده است، حاوی تمامی ساختارهای کلیدی زبان جاوا بوده و بنابراین مطمئن هستیم که اگر بتوانیم هر تحلیلی برای این زبان ارائه دهیم، به راحتی قادر خواهیم بود که آن را به کل زبان جاوا گسترش دهیم. زبان ارائه شده در این بخش از پایان‌نامه همان ConcurrentJava [۳۲] است که ساختار تبدیل نوع از ClassicJava [۴۲] به آن اضافه شده است. این نحو و تمام تحلیل‌ها و سیستم‌های نوع وابسته به آن بعداً در اثبات درستی

سیستم کشف رقابت داده گسترش می‌یابد. سیستم نوع پایه‌ای برای این زبان را در پیوست اول ارائه داده‌ایم. هدف نهایی از ارائه این بخش تعریف دقیق گراف جریان کنترل برای نمایش برنامه‌های زبان جاوای همروند مورد بررسی با استفاده از گراف‌های جهت‌دارد است.

۱.۲.۴ نحو مجرد

نحو مجرد زبان ConcurrentJava$^+$ را مطابق شکل ۲.۴ تعریف می‌کنیم. همان‌طور که مشاهده می‌شود یک برنامه دنباله‌ای از کلاس‌ها به همراه یک عبارت آغازین در انتها است. عبارت آغازین عبارتی است که برای اجرای یک برنامه ارزیابی شده، و نتیجه آن به عنوان نتیجه برنامه درنظر گرفته می‌شود. نتیجه ارزیابی هر عبارت در این زبان یک شئ یا مقدار ویژه `null` است.

P	::=	$defn^*\ e$	برنامه
$defn$	::=	`class` c `extends` $c\ \{field^*\ meth^*\}$	تعریف کلاس
$field$	::=	$t\ fd$	تعریف فیلد
$meth$	::=	$t\ md(par^*)\{e\}$	تعریف متد
par	::=	$t\ var$	تعریف پارامتر
e	::=	`new` t | var | `null` | $e.fd$ | $e.fd = e$	
		| $e.md(e^*)$ | $(t)e$ | `let` $var = e$ `in` e	
		| `synchronized` e `in` e | e.`start()`	عبارت‌ها
t	::=	c	اسم نوع
var	$\in$	$Identifiers \cup \{$`this`$\}$	نام متغیرها
c	$\in$	$Identifiers \cup \{$`Object`$\}$	نام کلاس‌ها
fd	$\in$	$Identifiers$	نام فیلدها
md	$\in$	$Identifiers \cup \{$`run`$\}$	نام متدها

شکل ۲.۴: نحو مجرد زبان ConcurrentJava$^+$

برنامه‌سازی همروند در این زبان با فراهم آوری ساختار e.`start()` پشتیبانی می‌شود که برای ایجاد یک ریسه جدید مورد استفاده قرار می‌گیرد. ارزیابی عبارت ایجاد ریسه جدید به این صورت است که ابتدا زیر عبارت e ارزیابی می‌شود (که نتیجه آن باید یک شئ باشد)، سپس فراخوانی متد بدون پارامتر `run` (با نوع برگشتی `Object`) بر روی آن شئ در یک ریسه جدید انجام می‌شود. این ریسه به محض ایجاد شدن به موازات ریسه والد اجرا می‌شود. عبارات ایجاد ریسه صرفاً به دلیل اثری که اجرا ریسه دارد مورد استفاده قرار می‌گیرد، و مقدار حاصل از ارزیابی خود عبارت همواره برابر با `null` است. در این زبان از قفل‌ها برای همگام‌سازی بین ریسه‌ای استفاده شده است. همانند زبان برنامه‌سازی جاوا هر شئ دارای یک قفل منحصر به فرد است که می‌تواند در دو حالت باز یا بسته باشد. رفتار عبارت `synchronized` e_1 `in` e_2 مشابه رفتار ساختار همگام‌سازی `synchronized` در جاوا است: ابتدا عبارت e_1 ارزیابی می‌شود، که باید یک شئ را نتیجه بدهد، سپس قفل منتسب به این شئ اخذ شده (یعنی بسته می‌شود) و عبارت e_2 ارزیابی می‌شود. هنگامی که ارزیابی عبارت e_2 به پایان

رسید (یعنی به یک شئ یا مقدار ویژه `null` ارزیابی شد) قفل شئ یاد شده آزاد می‌گردد (یعنی باز می‌شود) و نتیجه ارزیابی زیرعبارت $e_۲$ به عنوان نتیجه ارزیابی کل عبارت در نظر گرفته می‌شود. در زمانی که ریسه جاری عبارت $e_۲$ را ارزیابی می‌کند، گوییم، ریسه جاری قفل شئ یاد شده را در دست دارد. هر ریسه دیگر که برای اخذ این قفل تلاش کند مادامی که قفل مورد نظر آزاد نشده است بلوکه می‌شود. توجه کنید که ریسه‌های ایجاد شده قفل‌های اخذ شده توسط والد خود را به ارث نمی‌برند.

با داشتن نحو زبان، می‌توان تعدادی از مجموعه‌هایی که در ابتدای این فصل معرفی کردیم را به‌صورت دقیق تعریف کنیم. مجموعه $Methods$ مجموعه اسامی کامل متدها نام دارد و به‌صورت زیر قابل تعریف است:

$$Methods = \{c.md \mid c \in Identifiers \wedge md \in Identifiers\},$$

همان‌طور که مشاهده می‌شود اسامی کامل متدها رشته‌هایی به فرم $c.md$ است که در آن c شناسه کلاس و md شناسه متد است.

در ادامه به تابع $bodyOf$، که با دریافت نام کامل یک متد اعلان متد مورد نظر در برنامه $P_\star$ را برمی‌گرداند، نیاز خواهیم داشت. این تابع را می‌توان به‌صورت زیر تعریف کرد:

$$bodyOf(c.md) = \begin{cases} \Lambda & ; c = \texttt{Object}, \\ t\ md(par^*)\{e\} & ; P_\star; \emptyset \vdash \texttt{class}\ c\ \dots\ \{\dots t\ md(par^*)\{e\} \dots\}, \\ bodyOf(c'.md) & ; P_\star; \emptyset \vdash c <: c' \text{ در غیر این‌صورت اگر} \end{cases}$$

لازم به یادآوری است که در این پایان‌نامه نماد Λ برای نشان دادن رشته تهی، عملگر $\in$ برای بررسی وجود یک زیر عبارت در داخل یک عبارت، و «...» برای نشان دادن بخش‌هایی از یک رشته که مقدار دقیق آن برای ما اهمیت ندارد به‌کار می‌رود. توجه کنید که این تابع بدنه آن متد md را بر می‌گرداند که در کلاس c به‌صورت صریح اعلان شده یا به ارث برده شده است را بر می‌گرداند. همان‌طور که مشاهده می‌شود، این تابع را برحسب حکم‌های $P; E \vdash defn$ و $P; E \vdash t_۱ <: t_۲$ که در پیوست اول پایان‌نامه ارائه شده است، تعریف کرده‌ایم.

در این پایان‌نامه همچنین نیاز به تعریف دقیق چند مجموعه داریم که در همین‌جا به تعریف آن‌ها می‌پردازیم. مجموعه تمام اسامی کامل متدها در یک برنامه مانند $P_\star$ را با $Methods_\star$ نشان داده و به‌صورت زیر تعریف می‌کنیم:

$$Methods_\star = \{c.md \mid P_\star \vdash_\in (c, md)\},$$

که در آن حکم $P_\star \vdash_\in (c, md)$ با استفاده از سیستم زیر قابل اثبات است:

$$(۱)\ \frac{\texttt{class}\ c\ \ldots\ \{\ldots\ md(\ldots)\{\ldots\}\ \ldots\} \in P_\star}{P_\star \vdash_\in (c, md)}, \quad (۲)\ \frac{\begin{array}{c} P_\star \vdash_\in (c', md) \\ \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots meth_{۱\ldots n}\} \in P_\star \\ meth_i = t\ md'(\ldots)\{\ldots\} \implies md \neq md' \\ i \in ۱\ldots n \end{array}}{P_\star \vdash_\in (c, md)}.$$

مجموعه‌های $Fields$ و $Variables$ به ترتیب حاوی تمامی شناسه‌های فیلدها و متغیرها است. این مجموعه‌ها زیر مجموعه‌هایی محض از مجموعه شناسه‌ها هستند و می‌توان به‌صورت زیر تعریف کرد:

$$Fields \subset Identifiers,$$
$$Variables \subset Identifiers.$$

مجموعه شناسه فیلدها و شناسه متغیرهای ظاهر شده در یک برنامه $P_\star$ را به ترتیب با استفاده از $Fields_\star$ و $Variables_\star$ نشان می‌دهیم.

۲.۲.۴ گراف جریان کنترل

در تحلیل‌های جریان داده معمولاً یک برنامه را با استفاده از یک گراف جهت‌دار به نام گراف جریان کنترل نمایش می‌دهند. در این گراف گره‌ها نشان دهنده بلوک‌های ابتدایی برنامه و کمان‌ها نشان دهنده چگونگی انتقال جریان کنترل از بلوکی به بلوک دیگر است. دستوالعمل‌ها، عبارات، یا هر ساختار زبانی که رفتار آن‌ها در یک تحلیل جریان داده بررسی می‌شود بلوک ابتدایی نام دارد. به‌طور معمول فرض می‌شود که بلوک‌های ابتدایی برنامه مقصد (برنامه‌ای که در حال تحلیل آن هستیم) به‌صورت یکتا برچسب خورده‌اند. بنابراین، هر بلوک ابتدایی را می‌توان با استفاده از یک و تنها یک برچسب شناسایی کرد. معناشناخت هر زبانی ترتیبی برای ارزیابی عبارات و اجرای دستوالعمل‌ها مشخص می‌کند. با استفاده از این اطلاعات می‌توان بلوک‌های ابتدایی را به کمک کمان‌هایی به هم متصل کرد، به این ترتیب نمایشی از برنامه‌ی مقصد (سازگار با جریان کنترل در زمان اجرا) به دست می‌آید.

بدون از دست دادن کلیت مسأله می‌توان فرض کرد که یک برنامه تشکیل شده از گروهی از کلاس‌ها است (عبارت آغازین برنامه را می‌توان به عنوان بدنه یک متد از یک کلاس تصور کرد). بنابراین، در تمامی تحلیل‌هایی که برای انجام آن‌ها نیاز به گراف جریان کنترل داریم، این گراف را برای یک متد از یک برنامه در نظر می‌گیریم. در تعریف ۱.۴ گراف جریان کنترل را که در تحلیل‌های بین‌رویه‌ای پیمانه‌ای مورد استفاده قرار می‌گیرند ارائه داده‌ایم.

تعریف ۱.۴. برای یک برنامه مانند $P_\star$ و یک متد مانند $c.md$ در $Methods_\star$ گراف جریان کنترل بین‌رویه‌ای به‌صورت زیر تعریف می‌شود:

$$ICFG^{P_\star}_{c.md} = (B, F),$$

که در آن مؤلفه‌های آن به ترتیب گره‌ها و کمان‌ها (جریان‌ها) برای متد مورد نظر است، و به‌صورت زیر قابل تعریف هستند:

$$B = labelsOf(c.md, \emptyset),$$
$$F = flowsOf(c.md, \emptyset),$$

به قسمی که داریم:

$$labelsOf(m, cs) = \begin{cases} \emptyset & ; m \in cs, \\ labels(bodyOf(m)) & \\ \quad \cup (\bigcup \{labelsOf(m', cs \cup \{m\}) \mid & \\ \quad\quad m' \in mayCall(m) \wedge (m, m') \in SCC\}) & ; o.w. \end{cases}$$

$$flowsOf(m, cs) = \begin{cases} \emptyset & ; m \in cs, \\ flows(bodyOf(m)) & \\ \quad \cup (\bigcup \{flowsOf(m', cs \cup \{m\}) \mid & \\ \quad\quad m' \in mayCall(m) \wedge (m, m') \in SCC\}) & ; o.w. \end{cases}$$

■

همان‌طور که مشاهده می‌شود، در این تعریف از یک مجموعه به نام SCC استفاده کرده‌ایم. این مجموعه، که تعریف آن را در ادامه این بخش مشاهده می‌کنیم، حاوی تمام زوج اسامی متدهایی است که با همدیگر را به‌صورت بازگشتی فراخوانی می‌کنند. در ادامه این بخش به تعریف دقیق سایر توابع و مجموعه‌های مورد استفاده در این تعریف می‌پردازیم.

برچسب گذاری

مجموعه $Labels$ را به عنوان مجموعه‌ای شمارا از برچسب‌ها در نظر می‌گیریم. ماهیت دقیق این برچسب‌ها برای ما اهیمتی ندارد. بنابراین، این مجموعه را می‌توان مجموعه‌ای مانند مجموعه اعداد طبیعی تعریف کرد. در این پایان‌نامه، هر یک از اعضای این مجموعه را با نماد ℓ (احتمالاً با یک اندیس برای تفکیک چندین برچسب) نشان می‌دهیم تا ماهیت دقیق برچسب‌ها، به منظور اجتناب از پیچیدگی، مخفی بماند. با این توضیحات داریم:

$$Labels = \{\ell_{\cdot}, \ell_{۱}, \ell_{۲}, \dots\}.$$

عبارت‌ها و اعلان متدها در یک برنامه را به‌صورت زیر برچسب‌گذاری می‌کنیم. لازم به یادآوری است که برچسب‌های به‌کار رفته در یک برنامه باید دو به دو با هم متمایز باشند. به عبارتی دقیق‌تر، هدف از برچسب گذاری این است که هر

عبارت در یک برنامه فقط و فقط با برچسبی منحصر به فرد قابل شناسایی باشد.

$$t\ md(par^*)\ [\{]^{\ell_n}\ \ e\ \ [\}]^{\ell_x},$$

$$[\texttt{new}\ t]^{\ell},\qquad [var]^{\ell},\qquad [\texttt{null}]^{\ell},\qquad [e.fd]^{\ell},\qquad [e.fd=e]^{\ell},$$

$$[e.md(e^*)]^{\ell_c}_{\ell_r},\qquad [(t)e]^{\ell},\qquad [\texttt{let}\ var=e\ \texttt{in}\ e]^{\ell_n}_{\ell_x},$$

$$[\texttt{synchronized}\ e\ \texttt{in}\ e]^{\ell_n}_{\ell_x},\qquad [e.\texttt{start}()]^{\ell}.$$

توجه کنید که می‌توانستیم این برچسب‌گذاری را در نحو مجرد زبان نیز بیان کنیم، اما برای خواناتر شدن نحو زبان این کار را انجام نمی‌دهیم (در اصل نوشتن برنامه‌ای که یک رشته توصیف شده توسط شکل ۲.۴ را دریافت و آن را به‌صورت یکا برچسب گذاری می‌کند، کار ساده‌ای است). همچنین، توجه کنید که عبارات مربوط به فراخوانی توابع دارای دو برچسب متمایز هستند. این برچسب‌ها برای ساخت جریان‌های بین‌رویه‌ای مورد استفاده قرار می‌گیرند. دستورات همگام‌سازی و دستور تعریف متغیرهای محلی نیز دارای دو برچسب متمایز هستند. در مورد این دستورات، برچسب‌های جدا برای تفکیک دستور مورد نظر به دو بخش و بررسی جداگانه رفتار هر بخش مورد استفاده قرار می‌گیرد.

مجموعه $Flows$ حاوی جریان‌های درون‌رویه‌ای و بین‌رویه‌ای است. یک جریان در اصل یک دوتایی از برچسب‌ها است که جریان کنترل برنامه را مشخص می‌کند، اما برای ساخت تحلیل‌های بین رویه‌ای راحت‌تر است که جریان‌های بین‌رویه‌ای را از جریان‌ها درون‌رویه‌ای تفکیک کنیم. بنابراین، این مجموعه را به‌صورت زیر تعریف می‌کنیم که در آن جریان‌های درون‌رویه‌ای با زوج‌های مرتب از برچسب‌ها، و جریان‌های بین‌رویه‌ای با زوج برچسب‌هایی که با نقطه‌ویرگول از هم جدا شده‌اند نشان داده شده است:

$$Flows = \{(\ell_۱,\ell_۲) \mid \ell_۱,\ell_۲ \in Labels\} \cup \{(\ell_۱;\ell_۲) \mid \ell_۱,\ell_۲ \in Labels\}.$$

یک رشته برچسب‌دار از زبان را یک بلوک می‌گوییم. مجموعه $Blocks$ را به عنوان مجموعه تمام بلوک‌های ابتدایی تعریف می‌شود:

$$\begin{aligned}
Blocks = \{&[\texttt{new}\ t]^{\ell}, [var]^{\ell}, [\texttt{null}]^{\ell}, [e.fd]^{\ell}, [e.fd=e]^{\ell}, [(t)e]^{\ell} \mid \ell \in Labels\}\\
&\cup\{[e.md(e^*)]^{\ell_c}_{\ell_r} \mid \ell_c,\ell_r \in Labels\}\\
&\cup\{[e.\texttt{start}()]^{\ell} \mid \ell \in Labels\}\\
&\cup\{[begin(var_۱,\ldots,var_k)]^{\ell_n} \mid k \geq ۰ \wedge \ell_n \in Labels\}\\
&\cup\{[end(var_۱,\ldots,var_k)]^{\ell_x} \mid k \geq ۰ \wedge \ell_x \in Labels\}\\
&\cup\{[begin(var=e_۱;e_۲)]^{\ell_n} \mid \ell_n \in Labels\}\\
&\cup\{[end(var=e_۱;e_۲)]^{\ell_x} \mid \ell_x \in Labels\}\\
&\cup\{[enterMonitor(e_۱;e_۲)]^{\ell_n} \mid \ell_n \in Labels\}\\
&\cup\{[exitMonitor(e_۱;e_۲)]^{\ell_x} \mid \ell_x \in Labels\}.
\end{aligned}$$

تابع کمکی $init$: تابع $init$ با دریافت یک رشته برچسب ورودی آن را محاسبه می‌کند. با توجه به برچسب‌گذاری عبارت‌ها، تابع کمکی $init$ را می‌توان به‌صورت زیر تعریف کرد. توجه کنید که در این تابع و سایر توابع کمکی مجموعه $Prog$ به عنوان مجموعه تمام رشته‌های تولید شده با متغیرهای نحوی e و $meth$، تعریف می‌شود.

$$init : Prog \to Labels$$

$$\begin{aligned}
&init(t\ md(par^*)\ [\{]^{\ell_n}\ \ e\ \ [\}]^{\ell_x}) = \ell_n, && init([\texttt{new}\ t]^{\ell}) = \ell,\\
&init([var]^{\ell}) = \ell, && init([\texttt{null}]^{\ell}) = \ell,\\
&init([e.fd]^{\ell}) = init(e), && init([e_1.fd = e_2]^{\ell}) = init(e_1),\\
&init([e..md(e_1,\ldots,e_k)]^{\ell_c}_{\ell_r}) = init(e.), && init([(t)e]^{\ell}) = init(e),\\
&init([\texttt{let}\ var = e_1\ \texttt{in}\ e_2]^{\ell_n}_{\ell_x}) = init(e_1), && init([\texttt{synchronized}\ e_1\ \texttt{in}\ e_2]^{\ell_n}_{\ell_x}) = init(e_1),\\
&\multicolumn{2}{c}{init([e_1.\texttt{start}()]^{\ell}) = init(e_1).}
\end{aligned}$$

مجموعه برچسب‌های ورودی تمام عبارت‌ها و اعلان متدها در یک برنامه مانند $P_\star$ را با $init_\star$ نشان می‌دهیم.

تابع کمکی $final$: تابع $final$ با دریافت یک رشته مجموعه برچسب‌های خروجی آن را محاسبه می‌کند. مشابه تابع $init$، تابع کمکی $final$ به‌صورت زیر تعریف می‌شود:

$$final : Prog \to \mathcal{P}(Labels)$$

$$\begin{aligned}
&final(t\ md(par^*)\ [\{]^{\ell_n}\ \ e\ \ [\}]^{\ell_x}) = \ell_x, && final([\texttt{new}\ t]^{\ell}) = \ell,\\
&final([var]^{\ell}) = \ell, && final([\texttt{null}]^{\ell}) = \ell,\\
&final([e.fd]^{\ell}) = \ell, && final([e.fd = e]^{\ell}) = \ell,\\
&final([e.md(e^*)]^{\ell_c}_{\ell_r}) = \ell_r, && final([(t)e]^{\ell}) = \ell,\\
&final([\texttt{let}\ var = e\ \texttt{in}\ e]^{\ell_n}_{\ell_x}) = \ell_x, && final([\texttt{synchronized}\ e\ \texttt{in}\ e]^{\ell_n}_{\ell_x}) = \ell_x,\\
&\multicolumn{2}{c}{final([e.\texttt{start}()]^{\ell}) = \ell.}
\end{aligned}$$

مجموعه برچسب‌های خروجی تمام عبارت‌ها و اعلان متدها در یک برنامه مانند $P_\star$ را با $final_\star$ نشان می‌دهیم.

تابع کمکی $flows$: تابع $flows$ برای محاسبه جریان‌های کنترلی عبارت‌ها، اعم از درون‌رویه‌ای و بین‌رویه‌ای مورد استفاده قرار می‌گیرید. این تابع را به‌صورت زیر تعریف می‌کنیم:

$$flows : Prog \to \mathcal{P}(Flows)$$

$$
\begin{aligned}
flows(t\ md(par^*)\ [\{]^{\ell_n}\ \ e\ \ [\}]^{\ell_x}) &= flows(e)\\
&\cup\{(\ell_n, init(e)), (final(e), \ell_x)\},\\
flows([\texttt{new}\ t]^{\ell}) = \emptyset, \qquad flows([var]^{\ell}) &= \emptyset, \qquad flows([\texttt{null}]^{\ell}) = \emptyset,\\
flows([e.fd]^{\ell}) &= flows(e) \cup \{(final(e), \ell)\},\\
flows([e_1.fd = e_2]^{\ell}) &= flows(e_1) \cup flows(e_2)\\
&\cup\{(final(e_1), init(e_2)), (final(e_2), \ell)\}.
\end{aligned}
$$

جریان‌های کنترلی که بین متدها در اثر فراخوانی و برگشت از آن‌ها ایجاد می‌شود، جریان‌های بین‌رویه‌ای نام دارند. عبارت‌های فراخوانی متدها جایی است که جریان‌های بین‌رویه‌ای ایجاد می‌شود. توجه کنید، از آنجایی که هدف این پایان‌نامه ارائه تحلیل‌های پیمانه‌ای است، جریان‌های بین‌رویه‌ای قدری متفاوت‌تر از معمول تعریف می‌شود.

$$
\begin{aligned}
flows([e..md(e_1, \ldots, e_k)]^{\ell_c}_{\ell_r}) &= (\bigcup\{flows(e_i) \mid 0 \leq i \leq k\})\\
&\cup \left(\bigcup_{i=0}^{k-1}\{(final(e_i), init(e_{i+1}))\}\right)\\
&\cup\{(final(e_k), \ell_c)\} \cup IF,
\end{aligned}
$$

که در آن مجموعه IF به‌صورت زیر تعریف می‌شود:

$$IF = \{(\ell_c, \ell_n), (\ell_x, \ell_r)\},$$

به‌طوری که به ازای هر m' در $callees(\ell_c)$ که $(c.md, m') \in SCC$ داریم:

$$bodyOf(m') = t\ md(par_1, \ldots, par_k)\ \ [\{]^{\ell_n}\ \ e_b\ \ [\}]^{\ell_x}.$$

طبق این تعریف جریان‌های بین‌رویه‌ای از یک نقطه فراخوانی به نقطه ورودی تمامی متدهایی که توسط نقطه فراخوانی مورد نظر، احتمالاً، فراخوانی می‌شوند و متد فراخوانی کننده (متد $c.md$ که گراف جریان کنترل را برای آن می‌سازیم) را به‌صورت بازگشتی فراخوانی نمی‌کند، موجود است.

تابع $flows$ به ازای بقیه ساختارهای زبان به‌صورت زیر تعریف می‌شود:

$$
\begin{aligned}
flows([(t)e]^{\ell}) &= flows(e) \cup \{(final(e), \ell)\},\\
flows([\texttt{let}\ var = e_1\ \texttt{in}\ e_2]^{\ell_n}_{\ell_x}) &= flows(e_1) \cup flows(e_2)\\
&\cup\{(\ell_n, init(e_2)), (final(e_1), \ell_n), (final(e_2), \ell_x)\},\\
flows([\texttt{synchronized}\ e_1\ \texttt{in}\ e_2]^{\ell_n}_{\ell_x}) &= flows(e_1) \cup flows(e_2)\\
&\cup\{(\ell_n, init(e_2)), (final(e_1), \ell_n), (final(e_2), \ell_x)\}.\\
flows([e.\texttt{start}()]^{\ell}) &= flows(e) \cup \{(final(e), \ell)\}
\end{aligned}
$$

همان‌طور که مشاهده می‌شود، با توجه به این‌که فرض کرده‌ایم برنامه‌های ورودی خالی از خطای شروع مجدد ریسه هستند،

هیچ کمان بین‌رویه‌ای از مکان‌های ایجاد ریسه خارج نمی‌شود. علت عدم تعریف جریان‌های کنترلی بین رویه‌ای در این مکان این است که متدهایی که بدنه ریسه‌ها را تشکیل می‌دهند، با توجه به فرض خالی بودن برنامه‌های ورودی از خطای شروع مجدد ریسه، با هیچ متد دیگر برنامه SCC نیستند.

مجموعه جریان‌های کنترلی تمام عبارت‌ها و اعلان متدها در یک برنامه مانند $P_\star$ را با $flows_\star$ نشان می‌دهیم.

تابع کمکی $blocks$: تابع $blocks$ با دریافت یک رشته مجموعه بلوک‌های تشکیل دهنده آن را محاسبه می‌کند. این تابع را به‌صورت زیر تعریف می‌کنیم:

$$blocks : Prog \to \mathcal{P}(Blocks)$$

$$\begin{aligned}
&blocks(t\ md(t_1\ var_1, \ldots, t_k\ var_k)\ [\{]^{\ell_n}\ \ e\ \ [\}]^{\ell_x}) = \\
&\qquad \{[begin(var_1, \ldots, var_k)]^{\ell_n}, [end(var_1, \ldots, var_k)]^{\ell_x}\} \\
&\qquad \cup blocks(e), \\
&blocks([\texttt{new}\ t]^{\ell}) = \{[\texttt{new}\ t]^{\ell}\}, \qquad blocks([var]^{\ell}) = \{[var]^{\ell}\}, \\
&blocks([\texttt{null}]^{\ell}) = \{[\texttt{null}]^{\ell}\}, \qquad blocks([e.fd]^{\ell}) = blocks(e) \cup \{[e.fd]^{\ell}\}, \\
&blocks([e_1.fd = e_2]^{\ell}) = blocks(e_1) \cup blocks(e_2) \cup \{[e_1.fd = e_2]^{\ell}\}, \\
&blocks([e..md(e_1, \ldots, e_k)]_{\ell_r}^{\ell_c}) = (\bigcup\{blocks(e_i) \mid 0 \le i \le k\}) \\
&\qquad \cup \{[e..md(e_1, \ldots, e_k)]_{\ell_r}^{\ell_c}\}, \\
&blocks([(t)e]^{\ell}) = blocks(e) \cup \{[(t)e]^{\ell}\}, \\
&blocks([\texttt{let}\ var = e_1\ \texttt{in}\ e_2]_{\ell_x}^{\ell_n}) = blocks(e_1) \cup blocks(e_2) \\
&\qquad \cup \{[begin(var = e_1; e_2)]^{\ell_n}, [end(var = e_1; e_2)]^{\ell_x}\}, \\
&blocks([\texttt{synchronized}\ e_1\ \texttt{in}\ e_2]_{\ell_x}^{\ell_n}) = blocks(e_1) \cup blocks(e_2) \\
&\qquad \cup \{[enterMonitor(e_1; e_2)]^{\ell_n}, [exitMonitor(e_1; e_2)]^{\ell_x}\}, \\
&blocks([e.\texttt{start}()]^{\ell}) = blocks(e) \cup \{[e.\texttt{start}()]^{\ell}\}.
\end{aligned}$$

مجموعه بلوک‌های تمام عبارت‌ها و اعلان متدها در یک برنامه مانند $P_\star$ را با $blocks_\star$ نشان می‌دهیم.

تابع کمکی $labels$: تابع کمکی $labels$ برای استخراج مجموعه تمام برچسب‌های موجود در یک عبارت مورد استفاده قرار می‌گیرد:

$$labels : Prog \to \mathcal{P}(Labels)$$

$$\begin{aligned}
labels(e) = &\{\ell \mid \exists e' \cdot [e']^{\ell} \in blocks(e)\} \\
&\cup \{\ell \mid \exists e', \ell' \cdot [e']_{\ell'}^{\ell} \in blocks(e)\} \\
&\cup \{\ell \mid \exists e', \ell' \cdot [e']_{\ell}^{\ell'} \in blocks(e)\}.
\end{aligned}$$

همان‌طور که مشاهده می‌شود، این تابع با دریافت یک عبارت برچسب دار تمامی برچسب‌های آن عبارت و زیرعبارت‌های آن عبارت را محاسبه می‌کند. مجموعه برچسب‌های تمامی عبارت‌ها و اعلان متدها در یک برنامه مانند $P_\star$ را با $labels_\star$ نشان می‌دهیم.

۳.۲.۴ گراف جریان کنترل برای تحلیل‌های پیمانه‌ای

تحلیل اشاره‌گر و یک نسخه دیگر از تحلیل اجرای مجرد ارائه شده در این پایان‌نامه هر متد را تنها یک بار تحلیل می‌کند و هیچ جریان بین‌رویه‌ای در گراف جریان کنترل این گونه از تحلیل‌ها وجود ندارد.

برای انجام این گونه از تحلیل‌ها توابع کمکی $init$، $final$، $blocks$، $labels$، و تمام بندهای تابع $flows$ به جز بندهای مربوط به فراخوانی متدها همچنان بدون تغییر باقی می‌مانند. تابع کمکی $flows$، برای بلوک‌های فراخوانی متدها، به‌صورت زیر بازتعریف می‌شود:

$$\begin{aligned} flows([e..md(e_1,\ldots,e_k)]^{\ell_c}_{\ell_r}) = (\bigcup\{flows(e_i) \mid 0 \le i \le k\}) \\ \cup \left(\bigcup_{i=1}^{k-1}\{(final(e_i), init(e_{i+1}))\}\right) \\ \cup\{(final(e_k), \ell_c)\} \end{aligned}$$

همان‌طور که مشاهده می‌شود، در این تعریف هیچ جریان بین‌رویه‌ای وجود ندارد (بدیهی است که تابع انتقال مربوط به برچسب ℓ_n باید یک تابع همانی باشد.)

در نهایت گراف جریان کنترل برای تحلیل‌های پیمانه‌ای را به‌صورت زیر تعریف می‌کنیم:

تعریف ۲.۴. برای یک برنامه مانند $P_\star$ و یک متد مانند $c.md$ در $Methods_\star$ گراف جریان کنترل پیمانه‌ای. به‌صورت زیر تعریف می‌شود:

$$MCFG^{P_\star}_{c.md} = (labels(c.md), flows(c.md)),$$

که در آن مؤلفه‌ها به ترتیب گره‌ها و کمان‌ها (جریان‌ها) برای متد مورد نظر هستند. ■

۴.۲.۴ اطلاعات چندگانگی برچسب‌ها

چندگانگی یک شئ یا ریسه مجرد تخمینی ایستای از تعداد نمونه‌های آن است. از آنجایی که مکان‌های ایجاد ریسه و نیز ایجاد شئ همانند بخش‌های دیگر متن برنامه برچسب‌گذاری شده است، می‌توان با توجه به اطلاعات چندگانگی برچسب مربوط به آن محل‌ها، تخمینی از تعداد شئ‌ها و ریسه‌های ایجاد شده توسط آن‌ها را به‌دست آورد. لازم به یادآوری است که این تعریف‌ها برای هر زبان ممکن است قدری متفاوت باشد. این تعریف‌ها برای زبان ConcurrentJava$^+$ در نظر گرفته شده‌اند.

مجموعه $Multiplicities$ را به‌صورت زیر تعریف می‌کنیم:

$$Multiplicities = \{1, *, ?\}.$$

در این مجموعه ۱ نشان دهنده چندگانگی مفرد، $*$ نشان دهنده چندگانگی بیش از یک است (برای مثال تعداد ریسه‌ها و شئ‌هایی که ممکن است در یک تابع بازگشتی ایجاد شوند)، و $?$ نشان دهنده چندگانگی غیر مشخص است (برای مثال تعداد ریسه‌ها و شئ‌هایی که در یک متد غیر بازگشتی کتابخانه‌ای ممکن است ایجاد شود.)

در این پایان‌نامه، هنگامی که راجع به یک برنامه خاص بحث می‌کنیم، منظور ما از مجموعه برچسب‌ها و شناسه‌ها،

مجموعه برچسب‌ها و شناسه‌هایی است که در آن برنامه ظاهر می‌شوند. تابع $getLabMult$ برای استخراج چندگانگی یک برچسب (و در نتیجه چندگانگی یک عبارت) از برنامه $P_\star$ مورد استفاده قرار می‌گیرد:

$$getLabMult : Labels_\star \to Multiplicities$$

$$getLabMult(\ell) = \begin{cases} ? & ; \exists c, c', md, e \cdot \neg isRecursive(c.md) \\ & \quad \wedge \left[\texttt{class } c \texttt{ extends } c' \{ \dots md(\dots) \{ \dots [e]^\ell \dots \} \dots \}\right] \in P_\star \\ * & ; \exists c, c', md, e \cdot isRecursive(c.md) \\ & \quad \wedge \left[\texttt{class } c \texttt{ extends } c' \{ \dots md(\dots) \{ \dots [e]^\ell \dots \} \dots \}\right] \in P_\star \\ ۱ & ; o.w. \text{ و } \ell \in labels_\star \end{cases}$$

در بدنه این تابع، بند اول چندگانگی یک عبارت در بدنه متدی غیر بازگشتی را تعیین می‌کند، حال آنکه بند دوم آن چندگانگی عبارت‌ها در بدنه متدهای بازگشتی را تعیین می‌کند، در نهایت بند آخر بدنه تابع چندگانگی زیر عبارت‌های عبارت اولیه برنامه را تعیین می‌کند.

برای هر متد گزاره $isRecursive$ را به‌صورت زیر تعریف می‌کنیم:

$$isRecursive(m) \iff \exists m' \cdot (m, m') \in SCC$$

که در آن SCC، مجموعه جفت متدهایی است که به‌صورت بازگشتی متقابل تعریف شده‌اند. این مجموعه با توجه به گراف فراخوانی برنامه قابل تعریف است:

$$SCC = \{(m, m') \mid \exists m., \dots, m_k \in Methods_\star \cdot m. = m \wedge m_k = m' \\ \wedge m. \in mayCall(m_۱) \wedge \dots \wedge m_{k-۱} \in mayCall(m_k) \wedge m_k \in mayCall(m.)\}$$

در مورد چندگانگی ریسه‌ها و شئ‌ها، مشاهدات زیر را در نظر می‌گیریم:

- هر شئ یا ریسه‌ای که توسط یک ریسه چندگانه، یک متد بازگشتی، یا یک حلقه تکرار (در این زبان مورد بررسی قرار نمی‌گیرد) ایجاد شده باشد، دارای چندگانگی $*$ است.
- اگر شئ ریسه‌ای دارای چندگانگی $*$ باشد، آن ریسه دارای چندگانگی $*$ است.
- اگر چندگانگی محل ایجاد ریسه یا شئ‌ای $*$ باشد، ریسه یا شئ مورد نظر دارای چندگانگی $*$ است.
- ریسه ویژه `main` (ریسه اجرا کننده عبارت آغازین برنامه)، دارای چندگانگی ۱ است.
- چندگانگی ریسه‌ها و شئ‌هایی که در متدهای غیر بازگشتی ایجاد می‌شوند برابر با ? است.

این مشاهدات، در نهایت منجر به تعریف عملگر $\otimes$ می‌شود که در ادامه برای تعیین چندگانگی ریسه‌ها و شئ‌ها با توجه چندگانگی برچسب عبارت ایجاد کننده آن‌ها و نیز چندگانگی ریسه والد مورد استفاده قرار می‌گیرد. این عملگر را مطابق جدول ۱.۴ تعریف می‌کنیم.

جدول ۱.۴: عملگر $\otimes$

$\otimes$	۱	$*$	?
۱	۱	$*$	۱
$*$	$*$	$*$	$*$
?	۱	$*$	?

۳.۴ تحلیل اجرای مجرد و اطلاعات همروندی

یک رخداد را می‌توان به عنوان گذار (آنی) حالت سیستم که توسط یک اجرا از یک دستورالعمل به وقوع می‌پیوندد تعریف کرد. بنابراین، یک رخداد در اصل نمونه‌ای از اجرای یک دستورالعمل است. طبق این تعبیر برای یک دستورالعمل i می‌توان مجموعه‌ای شمارا از رخدادها را در نظر گرفت، که ما این مجموعه را چنین تعریف می‌کنیم:

$$[\![i]\!] = \{e_{\cdot}, e_{۱}, \dots\},$$

به‌طوری که e_j نشان دهنده j-امین نمونه از اجرای دستورالعمل i است. ما به چنین رخدادهایی عینی نیز می‌گوییم، چرا که متناظر با گذار حالت سیستم در اثر هر بار اجرای یک دستورالعمل در زمان اجرا هستند.

توجه کنید که فرض کرده‌ایم که دستورالعمل‌ها برحسب مکان منحصر به فرد آن‌ها در متن برنامه‌ها از هم متمایز شده‌اند. به عبارتی بهتر فرض ما بر این است که تمامی دستورالعمل‌های یک برنامه، برحسب مکان منحصر به فرد آن‌ها در متن آن برنامه، با استفاده از برچسب‌هایی، به‌صورت منحصر به فرد، برچسب خورده‌اند و دو دستوالعمل فقط به دلیل این‌که دارای برچسب‌های متمایزی هستند به عنوان دو دستورالعمل متفاوت شناخته می‌شوند. بنابراین، بدون از دست دادن کلیت مسأله، می‌توان فرض کرد که مجموعه رخدادهای عینی مربوط به دو دستورالعمل متمایز هیچ اشتراکی با هم ندارند.

به‌طور معمول رخدادها را می‌توان، در حالت کلی، با یک چندتایی به‌صورت صوری زیر بیان کرد:

$$\kappa(id, \tau, \rho),$$

که در آن κ نوع رخداد (مثلاً خواندن، نوشتن، اخذ، و یا آزادسازی)، id شناسه رخداد که برای تفکیک نمونه‌های مختلف اجرای یک دستورالعمل بخصوص به‌کار می‌رود، τ نشان دهنده شناسه ریسه‌ای است که باعث به وجود آمدن رخداد مورد نظر شده، و در نهایت ρ نشان دهنده منبعی است که رخداد به آن مربوط می‌شود (مثلاً در یک رخداد از نوع نوشتن، ρ می‌تواند یک مکان حافظه باشد.)

همان‌طور که پیش‌تر نیز اشاره کردیم یک برنامه توصیفی از مجموعه‌ای از اجراها بوده، و یک اجرا دنباله‌ای (احتمالاً

متناهی) از رخدادها است. در این پایان‌نامه ما اجراها را به عنوان دنباله‌ای از رخدادها در نظر نمی‌گیریم، بلکه با پیروی از [۴۳] یک اجرا را، به‌صورت معادل، به عنوان یک دوتایی از مجموعه تمام رخدادهایی که در آن اجرا رخ داده‌اند، و یک رابطه پیش‌رویدادی بر روی آن مجموعه (مؤلفه اول) تعریف می‌کنیم:

$$(E, \rightarrow).$$

توجه کنید که این نمایش تمام اطلاعات مفیدی که تعریف شهودی اجرا منتقل می‌کند را در بر دارد (می‌دانیم که ترتیب فیزیکی رخدادها در بسیاری از کاربردها اهمیتی ندارد).

اجرای $(E', \rightarrow')$ را پیشوند اجرای $(E, \rightarrow)$ گوییم، هرگاه $E' \subseteq E$ و $\rightarrow' \subseteq \rightarrow$ باشد. با وجود اینکه مفهوم پیشوند بودن یک اجرا از اجرایی دیگر تعبیری شهودی ندارد، اما ابزاری مناسب برای ارائه مفاهیم مهم‌تری است که در ادامه مشاهده می‌کنیم. با توجه به وجود برچسب‌های یکتا برای دستورالعمل‌های یک برنامه، هر دستورالعمل یک نقطه متمایز از متن آن برنامه را مشخص می‌کند. برای یک دستورالعمل مانند i از یک برنامه یک اجرا از برنامه مورد نظر تا آن دستورالعمل (نقطه‌ای از متن برنامه که i در آن قرار دارد) را به عنوان تمام پیشوندهای $(E', \rightarrow')$ ممکن از یک اجرا از آن برنامه تعریف می‌کنیم، که با فرض $E = E' \cap [\![i]\!]$، برای هر e در مجموعه E و برای هر e' در مجموعه E' داشته باشیم $e \not\rightarrow' e'$. به بیانی دیگر، یک اجرا از یک برنامه تا یک نقطه مشخص از متن آن برنامه (مثلاً دستورالعمل i) عبارت است از مجموعه تمام پیشوندهایی از یک اجرای برنامه مورد نظر که در آن هر ریسه تنها یک رخداد از دستورالعمل موجود در نقطه مورد نظر (یعنی i) را تولید کرده است، به‌طوری که این رخداد آخرین رخداد از دنباله رخدادهایی باشد که ریسه مورد نظر تولید می‌کند (به عبارتی دیگر ریسه مورد نظر بعد از آن رخداد، رخداد دیگری را تولید نکند).

تحلیل همروندی یا همان تحلیل ممکن‌است‌همزمان‌رخ‌دهند یک برنامه همروند، همان‌طور که از نامش مشخص است یک تحلیل با قطعیت may بوده، و هدف آن تشخیص ایستای جفت دستورالعمل‌هایی از برنامه مورد نظر است که رخداد یا رخدادهایی از هر کدام حداقل در یک اجرا رخ می‌دهند، به‌طوری که هیچ‌کدام با دیگری رابطه پیش‌رویدادی نداشته باشد. با توجه به مفاهیمی که تا کنون توسعه داده‌ایم، به‌طور آرمانی، جفت دستورالعمل‌های $(i_۱, i_۲)$ در نتیجه تحلیل مشاهده می‌شود هرگاه اجرایی مانند $(E, \rightarrow)$ موجود باشد، به طوری‌که با فرض $E_۱ = E \cap [\![i_۱]\!]$ و $E_۲ = E \cap [\![i_۲]\!]$ بتوانیم نشان دهیم که $E_۱ \neq \emptyset$، $E_۲ \neq \emptyset$، و رخدادهای $e_۱$ و $e_۲$، به ترتیب، در مجموعه‌های $E_۱$ و $E_۲$ وجود دارند به‌قسمی که $e_۱ \not\rightarrow e_۲$ و $e_۲ \not\rightarrow e_۱$.

تحلیل همروندی ارائه شده در این پایان‌نامه ریزدانه‌تر از تحلیل‌های همروندی مرسوم است، چرا که این تحلیل به رخدادها، که نمونه‌های اجرای دستورالعمل‌ها هستند، مربوط می‌شود. تحلیل مورد نظر یک تحلیل مبتنی بر جریان داده، حساس به جریان کنترل، حساس به زمینه، و با قطعیت may است. این تحلیل برای هر نقطه از یک برنامه مجموعه تمام رخدادهایی که ممکن (با توجه به تمام اجراهای ممکن) است در اثر اجرای آن برنامه تا نقطه مورد نظر رخ دهند را مشخص می‌کند. این تحلیل، همچنین، تخمینی از رابطه پیش‌رویدادی بر روی این رخدادها را محاسبه می‌کند. البته رابطه پیش‌رویدادی و نیز شیوه نمایش رخدادها در پاسخ این تحلیل با معادل عینی آن‌ها متفاوت است.

رخدادهای محاسبه شده توسط این تحلیل، که ما آن‌ها را رخدادهای مجرد می‌نامیم، در اصل نشان دهنده مجموعه‌ای از رخدادهای عینی است. این پدیده که موجودیت‌های تشخیص داده شده توسط تحلیل‌های ایستا نشان دهنده مجموعه‌ای از موجودیت‌های در زمان اجرا باشد، در تحلیل‌های ایستای محاسبه‌پذیر (تحلیل‌هایی که حل معادلات آن‌ها برای یافتن جواب

با استفاده از یک الگوریتم قابل انجام است) مشاهده می‌شود. برای مثال، در یک تحلیل ایستای شکل هیپ یک برنامه تمامی اشیاء قابل ایجاد در یک محل ایجاد شئ با استفاده از یک شئ مجرد نشان داده می‌شوند. این شئ مجرد که متناظر با یک محل ایجاد شئ در برنامه (مثلاً دستور new در جاوا) است، نشان دهنده مجموعه تمام شئ‌های عینی است که، در تمامی اجراهای ممکن آن برنامه، توسط محل مورد نظر ایجاد می‌شوند. برای روشن‌تر شدن موضوع فرض کنید که برنامه زیر توسط چنین تحلیلی مورد بررسی قرار گرفته است (توجه کنید که بخش‌هایی از متن برنامه که برای ما اهمیت ندارند با استفاده از نقطه‌چین مشخص شده است، همچنین، خطوط برنامه را با برچسب‌هایی مشخص کرده‌ایم):

```
    ...
۱ : while (...) {
        ...
۲ :     x = new T();  //allocate a T object
        ...
۳ : }
    ...
```

از آنجایی که تعیین مقدار یک عبارت به‌صورت ایستا در حالت کلی ممکن نیست، تحلیل‌های ایستا به‌طور معمول فرض می‌کنند که بدنه حلقه‌های تکرار و توابع بازگشتی، به ازای یک عدد طبیعی دلخواه $n > ۰$، n-بار قابلیت اجرا دارند. همچنین، تمامی بندهای دستورالعمل‌های شرطی (مانند ساختارهای $\texttt{if} - \texttt{else}$ و ساختارهایی مانند $\texttt{switch} - \texttt{case}$ در زبان جاوا) ممکن است اجرا شوند. چنین فرض‌های محافظه‌کارانه‌ای است که، در صورت نیاز به ارائه تحلیلی محاسبه‌پذیر، ما را مجبور می‌کند که مجموعه تمام شئ‌های عینی قابل ایجاد توسط محل ایجاد شئ در خط شماره ۲ را به عنوان یک شئ مجرد، متناظر با این محل (یعنی مثلاً شئ مجرد شماره ۲)، در نظر بگیریم. به‌این ترتیب تمامی اشیاء قابل ایجاد در محل ایجاد شئ ۲ به‌صورت یک کل (همان شئ مجرد) در نظر گرفته می‌شوند. بنابراین، یک شئ مجرد متناظر با یک مکان ایجاد شئ در متن برنامه است (یک دستوالعمل برچسب دار) که در اصل شناسه‌ای برای مجموعه تمام شئ‌هایی است که ممکن است (با در نظر گرفتن تمامی اجراهای ممکن برنامه) در زمان اجرا توسط آن مکان ایجاد شوند. به‌طور مشابه مکان‌های ایجاد رخداد در برنامه‌ها دستورالعمل‌ها هستند. بنابراین، یک رخداد مجرد متناظر با یک مکان در متن برنامه است که در اصل شناسه‌ای برای مجموعه تمام رخدادهایی است که ممکن است در زمان اجرا توسط دستورالعمل موجود در آن نقطه ایجاد شوند.

تحلیل همروندی ارائه شده در این پایان‌نامه حساس به زمینه است، یعنی بین فراخوانی‌های متعدد یک رویه، و در نتیجه مجموعه رخدادهایی که در هر فراخوانی قابلیت ایجاد دارند، تفاوت قائل می‌شود. ما این کار را با پارامتریزه کردن رخدادهای مجرد انجام می‌دهیم. به این ترتیب که رخداد مجرد منتسب به دستورالعملی که در یک رویه قرار دارد، با شناسه رخداد، شناسه ریسه، و مکان مربوط به رخداد پارامتریزه شده است. بنابراین، رخداد مجرد منتسب به دستورالعمل مورد نظر در هر مکان فراخوانی رویه در بر گیرنده آن، متناسب با زمینه فراخوانی، نمونه‌سازی شده و یک رخداد مجرد متناسب با آن زمینه برای دستورالعمل یاد شده محاسبه می‌شود. در نتیجه‌ی انتساب رخدادهای مجرد پارامتریزه شده به هر دستورالعمل می‌توان گفت که یک نقطه از برنامه، علاوه بر رخداد پارامتریزه منتسب به آن، در عمل، با تعدادی (متناهی) رخداد مجرد

در ارتباط است، به‌طوری که هر یک از این رخدادها متناظر با یک مکان فراخوانی در متن برنامه است. بنابراین، برای یک دستورالعمل i در یک برنامه می‌توان مجموعه تمام رخدادهای مجرد مربوط به این دستورالعمل (متسب شده توسط تحلیل یا در اثر نمونه‌سازی در یک مکان فراخوانی) را به‌صورت زیر تعریف کرد:

$$\|i\| = \{\widehat{e}, \widehat{e}, \ldots, \widehat{e_n}\},$$

به‌طوری که $\widehat{e}$ رخداد منتسب به i توسط تحلیل بوده، و $\widehat{e_j}$ رخداد مجردی است که در j-امین مکان فراخوانی رویه در بر گیرنده دستورالعمل i، با نمونه‌سازی $\widehat{e}$ متناسب با زمینه جاری در مکان فراخوانی مورد نظر به‌دست آمده است. لازم به یاد آوری است که همین ایده پارامتریزه کردن رخدادهای مجرد است که اجازه تحلیل رویه‌ها به‌صورت جدا از هم و ارائه تحلیلی پیمانه‌ای را به ما می‌دهد.

علاوه بر محاسبه مجموعه رخدادهای مجرد برای هر نقطه از یک برنامه، هدف دیگر تحلیل همروندی ارائه شده در این پایان‌نامه محاسبه تخمینی از رابطه پیش‌رویدادی بر روی این رخدادها است. در صورتی که $\widehat{e_1}$ و $\widehat{e_2}$، رخدادهای مجردی باشند که عضو مجموعه رخدادهای محاسبه شده توسط تحلیل همروندی یاد شده بوده و، به ترتیب، مربوط به دستورالعمل‌های i_1 و i_2 در برنامه مورد بررسی باشند، رابطه پیش‌رویدادی محاسبه شده توسط این تحلیل، که با نماد $\mapsto$ نمایش می‌دهیم، حاوی دوتایی $(\widehat{e_1}, \widehat{e_2})$ است، هرگاه:

- نمونه‌های اجرای دستورالعمل‌های i_1 و i_2 توسط ترتیب برنامه مرتب شده‌اند، یا
- یک رخداد مجرد $\widehat{e_3}$ موجود است، به قسمی که $(\widehat{e_1}, \widehat{e_3})$ و $(\widehat{e_3}, \widehat{e_2})$.

همان‌طور که مشاهده می‌شود این تخمین بسیار غیر دقیق است، چرا که، بر خلاف رابطه پیش‌رویدادی، رابطه بین رخدادهای دو ریسه مختلف که با استفاده از عملگرهای همگام‌سازی بین‌ریه‌ای مرتب شده‌اند در نظر گرفته نشده است. این عدم دقت را بعداً با پردازش اطلاعات حاصل شده از تحلیل یاد شده جبران می‌کنیم. توجه کنید که در این پردازش نیاز به اطلاعات بیش‌تر (بیش‌تر از آن چیزی که تحلیل محاسبه کرده است) وجود ندارد.

۱.۳.۴ فضای اطلاعات تحلیل

در این بخش از پایان‌نامه مجموعه اجراهای مجرد را معرفی می‌کنیم. این مجموعه به همراه یک ترتیب جزئی که بر روی آن تعریف می‌شود، تشکیل یک مشبک تام را می‌دهد. تحلیل اجراهای مجرد معرفی شده در ادامه این فصل از این مجموعه به عنوان فضای اطلاعات تحلیل استفاده می‌کند.

پیش‌تر مشاهده کردیم که می‌توان نمایشی صوری از رخدادها ارائه داد که حاوی تمام اطلاعات مفید در مورد هر رخداد است. در این بخش ابتدا فضای رخدادها را به عنوان مجموعه‌ای تعریف می‌کنیم که وابسته به چند مجموعه دیگر است که آن‌ها را می‌توان متناسب با کاربرد به‌صورت اختصاصی تعریف کرد. چنین تعریف عمومی راه را برای به‌کارگیری ایده‌های ارائه شده در تحلیل همروندی این پایان‌نامه در زبان‌های برنامه‌نویسی مختلف و نیز استفاده از مفاهیم ارائه شده در این بخش در کاربردهایی به غیر از کشف رقابت داده هموار می‌کند.

تعریف ۱.۴. مجموعه رخدادها را به‌صورت زیر تعریف می‌کنیم:

$$Events = \{\kappa(id, \tau, \rho) \mid \kappa \in Kinds \wedge id \in EID \wedge \tau \in Threads \wedge \rho \in eventLoci(\kappa)\}$$

در این تعریف EID مجموعه‌ای شمارا از شناسه‌های رخدادها است. $Kinds$ مجموعه‌ای متناهی از نوع رخدادها است. همچنین، تابع $eventLoci$ مجموعه اطلاعاتی که می‌تواند به عنوان جزئیات در هر رخداد مورد استفاده قرار گیرد را مشخص می‌کند. با توجه به مدل حافظه زبان مورد استفاده، این جزئیات می‌تواند وابسته به نوع رخداد باشد. در نهایت مجموعه $Threads$ مجموعه‌ای شمارا از شناسه‌های ریسه‌ها است. ■

نکته. زمانی که نوع یک رخداد و یا هر یک از مؤلفه‌های آن (در بررسی‌ها نظری در پایان‌نامه) برای ما اهمیت نداشته باشد، از نوشتن آن‌ها خودداری می‌کنیم. از نماد «...» برای نشان دادن مؤلفه یا مؤلفه‌هایی که نوشته نشده‌اند استفاده می‌کنیم.

پیش‌تر در مورد ضرورت وجود هر یک از مؤلفه‌ها در یک رخداد بحثی کلی ارائه دادیم، در اینجا این ضرورت را در قالب نیازمندی کشف رقابت داده بیان می‌کنیم. همان‌طور که در تعریف رقابت داده و رخدادهای متضاد مشاهده کردیم، وجود اطلاعات نوع رخداد، ریسه ایجاد کننده آن، و نیز مکان حافظه‌ای که رخداد به آن مربوط می‌شود لازم و ضروری است. لزوم وجود مؤلفه شناسه رخدادها را نیز می‌توان به دو صورت توجیح کرد: (۱) با توجه به بحث‌هایی که در ابتدای این بخش مشاهده کردیم، id در اصل یک کدگذاری برای متن یک دستورالعمل و نوبت اجرای آن است، چرا که می‌دانیم هر رخداد در اصل نمونه‌ای از اجرای یک دستورالعمل بوده و هر دستورالعمل با توجه به برنامه و مکان منحصر به‌فردی از آن برنامه که در آن قرار دارد شناسایی می‌شود؛ (۲) در مورد نیازمندی کشف رقابت داده نبود این مؤلفه موجب چشم‌پوشی تحلیل‌ها از برخی از رخدادها می‌شود. برای مثال، برنامه نشان داده شده در شکل ۳.۴ متشکل از دو ریسه است که کد مربوط به این ریسه‌ها در دو ستون نوشته شده است. همان‌طور که مشاهده می‌شود، اگر رخدادها به‌صورت منحصر به‌فرد با یک شناسه مشخص نشوند، رخدادهای متناظر با خطوط ۱ و ۵ به اشتباه نسبت به هم مرتب در نظر گرفته می‌شوند. البته لازم به یاد آوری است که دلیل اول برای لزوم وجود مؤلفه id کافی بود، و دلیل دوم را صرفاً از دیدگاه مسأله کشف رقابت داده بیان کردیم.

ریسه ۱		ریسه ۲	
۱:	`i:=0;`	۵:	`i:=2;`
۲:	`lock(mylock);`	۶:	`lock(mylock);`
۳:	`i:=1;`	۷:	`i:=3;`
۴:	`unlock(mylock);`	۸:	`unlock(mylock);`

شکل ۳.۴: یک برنامه نمونه

تعریف ۲.۴. مجموعه تمام اجراهای مجرد را به‌صورت زیر تعریف می‌کنیم:

$$AE = \{(E, \mapsto) \mid E \subseteq Events \wedge \mapsto \subseteq E \times E\}$$

■

ترتیب جزئی $\sqsubseteq$ را بر روی مجموعه AE به‌صورت زیر تعریف می‌کنیم، که در آن η_1 و η_2 اعضای دلخواهی از AE هستند:

$$\eta_1 = (E_1, \mapsto_1), \eta_2 = (E_2, \mapsto_2) :$$
$$\eta_1 \sqsubseteq \eta_2 \iff E_1 \subseteq E_2 \wedge \mapsto_1 \subseteq \mapsto_2 .$$

با توجه به این ترتیب جزئی $\bot = (\emptyset, \emptyset)$ کوچکترین عنصر مجموعه AE بوده و کوچکترین کران بالا برای هر زیرمجموعه Y از AE به‌صورت زیر تعریف می‌شود:

$$\bigsqcup Y = (\bigcup\{E' \mid (E', \mapsto') \in Y\}, \bigcup\{\mapsto' \mid (E', \mapsto') \in Y\}).$$

حال با توجه به لم ۲.۲ در فصل ۲ نتیجه می‌گیریم که $(AE, \sqsubseteq, \bigsqcup, \bot)$ یک مشبک تام است. بنابراین، می‌توان آن را به عنوان فضای اطلاعات در یک تحلیل مبتنی بر چارچوب یکنوا استفاده کرد.

نکته. همان‌طور که در ادامه مشاهده خواهیم کرد، مجموعه AE برای یک برنامه در یک زبان خاص تعریف می‌شود. در حالتی که مجموعه AE متناهی است، ارتفاع مشبک (طول طویل‌ترین زنجیر در مجموعه مرتب جزئی) به راحتی با استفاده از اندازه مجموعه $Events$ قابل محاسبه است. در صورتی که $|Events| = n$ باشد، ارتفاع مشبک برابر با $n^2 + n$ است.

۲.۳.۴ اصلاح کننده نوع norace

در این بخش، حاشیه‌نویسی‌هایی برای اصلاح نوع‌ها ارائه می‌دهیم. نوع‌ها با افزودن اصلاح کننده `norace` به ابتدای اسم نوع قابل اصلاح هستند. در این بخش دو قاعده استنتاج برای گسترش سیستم نوع استاندارد، معرفی شده در پیوست اول، به منظور تعیین رابطه زیرنوعی در حضور این اصلاح کننده ارائه شده است.

نحو مجرد زبان جاوای همروند را به‌صورت نشان داده شده در شکل ۴.۴ گسترش می‌دهیم.

$meth$	::=	$t\ md(par^*)\ q\ \{\ e\ \}$	تعریف متد
par	::=	$t\ var$	پارامتر
t	::=	$q\ c$	نوع
q	::=	$(\texttt{norace})^?$	اصلاح کننده

شکل ۴.۴: متغیرهای نحوی گسترش یافته

قاعده خوش‌فرم بودن نوع‌ها: قاعده زیر را برای بررسی خوش فرم بودن نوع‌ها، به مجموعه قواعد استاندارد اضافه می‌کنیم:

$$\frac{P; E \vdash t \qquad \forall t' \cdot t \neq \texttt{norace}\ t'}{P; E \vdash \texttt{norace}\ t}$$

قاعده وارسی زیرنوع‌ها: فاعده زیر را برای وارسی زیرنوع بودن، به مجموعه قواعد استاندارد اضافه می‌کنیم:

$$\frac{P; E \vdash t_1 <: t_2}{P; E \vdash \texttt{norace}\ t_1 <: \texttt{norace}\ t_2}$$

۳.۳.۴ تحلیل بین‌رویه‌ای

در این بخش یک تحلیل اجرای مجرد محاسبه ناپذیر (نیم محاسبه‌پذیر)، اما ساده برای اثبات درستی، ارائه می‌دهیم. این تحلیل را با استفاده از زبان جاوای همروند و به عنوان نمونه‌ای از چارچوب یکنوا نمایش داده‌ایم. در ادامه با ارائه تحلیل اجرای مجرد دیگری هم مشکل محاسبه‌پذیر بودن و هم مشکل پیمانه‌ای بودن را حل می‌کنیم، و نشان می‌دهیم و در انتهای این بخش ثابت می‌کنیم که پاسخ این دو تحلیل با هم برابر است. علت اینکه از همان اول به‌صورت مستقیم از تحلیل پیمانه‌ای استفاده نکردیم این است که اثبات درستی برای چنین تحلیلی دشوار است. حال آنکه اثبات درستی برای یک تحلیل بین‌رویه‌ای و اثبات معادل بودن جواب‌های دو تحلیل (اولی برنامه را خالی از رقابت داده تشخیص می‌دهد اگر و تنها اگر دومی آن را خالی از رقابت داده تشخیص دهد) کاری به مراتب ساده‌تر است.

تحلیل اجرای مجرد را برای متدی مانند $c.md$ از برنامه $P_\star$ تعریف می‌کنیم. همان‌طور که پیش‌تر بیان کردیم، برای ارائه یک تحلیل که از مجموعه AE استفاده می‌کند نیاز داریم که مجموعه‌های EID، $Kinds$ و $Threads$، نیز و تابع $eventLoci$ به‌طوری که مطابق با زبان برنامه‌سازی مورد بررسی باشد نمونه‌سازی کنیم. این مجموعه‌ها را برای زبان جاوای همروند ارائه شده در این پایان‌نامه به‌صورت زیر تعریف می‌کنیم:

$$\begin{aligned}
&EID = \{s \mid s \in Labels_\star^+\},\\
&Kinds = \{R, W, N, X\},\\
&Threads = \{s^m \mid s \in TID^+ \wedge m \in Multiplicities\},\\
&eventLoci(\kappa) = \begin{cases} \mathcal{P}(Val \times Fields_\star) & ;\kappa \in \{R, W\} \\ \mathcal{P}(Val) & ;\kappa \in \{N, X\}. \end{cases}
\end{aligned}$$

که در آن مجموعه TID، مجموعه شناسه ریسه‌ها، به‌صورت زیر تعریف می‌شود:

$$TID = \{\texttt{main}, \tau_\cdot, \tau_1, \dots\},$$

به‌طوری که $\{\tau_\cdot, \tau_1, \dots\} \subseteq Labels_\star$ است. در مجموعه $Kinds$ نوع‌ها به ترتیب از چپ به راست مربوط به رخداد از نوع خواندن از حافظه، نوشتن در حافظه، ورود به مانیتور، و خروج از مانیتور است. در نهایت مجموعه $Multiplicities$، در برگیرنده اطلاعات چندگانگی ریسه‌ها و مکان‌های حافظه است که پیش‌تر در مورد آن بحث کرده‌ایم.

حال فرض می‌کنیم که $ICFG_{c.md}^{P_\star} = (B, F)$ گراف جریان کنترل بین‌رویه‌ای برای متد یاد شده باشد. تحلیل ارائه شده یک نمونه از چارچوب یکنوا است. بنابراین، می‌توان آن را با استفاده از یک شش‌تایی به‌صورت زیر بیان کرد:

$$\mathsf{AEA}^{\mathsf{int}} = (\mathcal{AE}, \mathcal{F}, F, E, \iota, f_\cdot),$$

که در آن $\mathcal{AE} = (AE, \sqsubseteq, \bigsqcup, \bot)$ یک مشبک تام فضای اطلاعات مورد نظر ما است. مجموعه $\mathcal{F}$ مجموعه‌ای از توابع یکنوا بر روی مجموعه AE است:

$$\mathcal{F} = \{f : AE \to AE, g : AE \times AE \to AE \mid f, g \text{ یکنوا هستند}\}.$$

مجموعه F همان مؤلفه دوم گراف جریان کنترل بین‌رویه‌ای است. همچنین، مؤلفه E به‌صورت مجموعه تک‌نقطه‌ای $\{init(bodyOf(c.md))\}$ تعریف می‌کنیم. چرا که تحلیل ارائه شده یک تحلیل رو به جلو است. مؤلفه ι را برابر با کوچکترین عنصر مشبک، یعنی $\bot$، قرار می‌دهیم. مؤلفه $f_.$ تابعی است که برچسب‌های موجود در $Labels_\star$ را به توابعی مناسب در مجموعه $\mathcal{F}$ نگاشت می‌کند:

$$f_. : Labels_\star \to \mathcal{F}$$

این تابع متناسب با عبارتی که برچسب مورد نظر برای برچسب‌زدن آن به‌کار رفته است، تابع انتقال مناسبی از داخل $\mathcal{F}$ انتخاب می‌کند. در ادامه با تعریف این تابع و دستگاه معادلات جریان داده تعریف تحلیل را تکمیل می‌کنیم.

در ارائه تحلیل اجرای مجرد مورد نظر با اطلاعات نوع زیرعبارات بدنه متد یاد شده نیاز داریم. در صورتی که بدنه متد $c.md$ به‌فرم زیر باشد:

$$bodyOf(c.md) = t_q\ md(t_{q_1}\ p_1, \ldots, t_{q_k}\ p_k)\ q.\{e'\}$$

محیط انتساب نوع برای این متد به‌صورت زیر تعریف می‌شود:

$$E_\star = \{\texttt{this} : q.\ c, p_1 : t_{q_1}, \ldots, p_k : t_{q_k}\}$$

توجه کنید که t_q و t_{q_1} تا t_{q_k} نوع‌های (احتمالاً) اصلاح شده هستند. در این تحلیل، محیط انتساب نوع در داخل بلوک‌های `let` به‌گونه‌ای به‌روزرسانی می‌شود که حاوی اطلاعات نوع مربوط به متغیر محلی تعریف شده در آن بلوک باشد. در حین خروج از هر بلوک `let` نیز متغیر اضافه شده به محیط انتساب نوع در اثر ورود به این بلوک حذف می‌شود. لازم به یادآوری است که محیط انتساب نوع $E_\star$ به گره آغازین گراف جریان کنترل متد مورد نظر منتسب شده است، و به هر گره دیگر در این گراف یک محیط انتساب نوع نسب داده شده است که برابر با اجتماع محیط‌های انتساب نوع مربوط به تمام گره‌هایی است که جریانی از آن‌ها وارد گره مورد نظر می‌شود. چنین محیط انتساب نوعی را محیط انتساب نوع وراثتی می‌گوییم. به این ترتیب می‌توان در هر گره به اطلاعات نوع هر زیرعبارت از بدنه متد دست پیدا کرد.

تحلیل اجرای مجرد معرفی شده در این پایان‌نامه، همان‌طور که پیش‌تر نیز به نحوی اشاره شده بود، یک تحلیل جریان داده رو به جلو و با قطعیت may است. دستگاه معادلات جریان داده که در آن $AEA^{\mathsf{int}}_\circ(\ell)$ و $AEA^{\mathsf{int}}_\bullet(\ell)$ به ازای $\ell \in Labels_\star$ متغیرهای مجهول دستگاه هستند. دستگاه یاد شده را برای این تحلیل را به‌صورت زیر تعریف می‌کنیم:

$$AEA^{\mathsf{int}}_\circ(\ell) = \bigsqcup\{AEA^{\mathsf{int}}_\bullet(\ell') \mid (\ell', \ell) \in F \vee (\ell'; \ell) \in F\} \qquad ; \ell \in B \tag{۱.۴}$$

همچنین، برای هر برچسب ℓ در مجموعه B که برچسب یک محل فراخوانی متد نیست داریم:

$$AEA^{\mathsf{int}}_{\bullet}(\ell) = f_{\ell}(AEA^{\mathsf{int}}_{\circ}(\ell)) \tag{۲.۴}$$

در نهایت معادلات جریان داده برای نقاط فراخوانی و ایجاد ریسه به‌صورت زیر تعریف می‌شود:

$$AEA^{\mathsf{int}}_{\bullet}(\ell_c) = f^{call}_{\ell_c}(AEA^{\mathsf{int}}_{\circ}(\ell_c)) \tag{۳.۴}$$

که در آن $\ell_c \in B$ بوده و یک برچسب ℓ_r در B موجود است به‌قسمی که $[e.md(e^*)]^{\ell_c}_{\ell_r} \in blocks_\star$.

$$AEA^{\mathsf{int}}_{\bullet}(\ell_r) = f^{ret}_{\ell_c,\ell_r}(AEA^{\mathsf{int}}_{\circ}(\ell_c), AEA^{\mathsf{int}}_{\circ}(\ell_r)) \tag{۴.۴}$$

که در آن $\ell_c, \ell_r \in B$ بوده به‌قسمی که $[e.md(e^*)]^{\ell_c}_{\ell_r} \in blocks_\star$.

تعریف ۳.۴ (پاسخ تحلیل بین‌رویه‌ای). زوج تابع $(AEA^{\mathsf{int}}_{\circ}, AEA^{\mathsf{int}}_{\bullet})$ که در معادلات (۱.۴) تا (۴.۴) برای یک متد $c.md$ در برنامه $P_\star$، و خلاصه متد ms صدق می‌کند، پاسخ تحلیل اجرای مجرد برای متد یاد شده نام دارد. این پاسخ را در قالب یک تابع به‌صورت زیر بیان می‌کنیم:

$$IntAnalyse(P_\star, c.md, ms) = \{(\circ, AEA^{\mathsf{int}}_{\circ}), (\bullet, AEA^{\mathsf{int}}_{\bullet})\}$$

■

قبل از تعریف توابع انتقال و تکمیل فرآیند تعریف تحلیل اجرای مجرد لازم است که مفهوم خلاصه متدها را معرفی کنیم. به زبان ساده، خلاصه متدها مکانی برای ذخیره پاسخ تحلیل متدهای تحلیل شده است.

تعریف ۴.۴ (خلاصه متد). خلاصه متدها، نگاشتی از نام کامل متدها به زیرمجموعه‌ایی از اجراهای مجرد که نشان دهنده پاسخ تحلیل برای متدهای مورد نظر در برچسب‌های پایانی آن‌ها است. به‌طور دقیق‌تر، این نگاشت را به‌صورت زیر تعریف می‌کنیم:

$$ms : Methods \to \mathcal{P}(AE).$$

هنگامی که $ms(c.md) = \emptyset$ باشد، به این معنا است که متد مورد نظر هنوز تحلیل نشده است، و در نتیجه خلاصه‌ایی برای آن موجود نیست. در غیر این‌صورت برای یک خلاصه متد ms' که برای هر m در مجموعه $mayCall(c.md)$ داریم $ms'(m) \neq \emptyset$، خلاصه متد متد $c.md$ به‌صورت زیر تعریف می‌شود:

$$ms(c.md) = \{IntAnalyse(P_\star, c.md, ms)(\bullet)(final(bodyOf(c.md)))\}$$

■

صورت کلی توابع انتقال برای تمام برچسب‌ها به غیر از آن‌هایی که برچسب یک محل فراخوانی متد هستند به‌صورت زیر

است:

$$f_\ell(\eta) = \eta \sqcup gen(\eta)([e]^\ell) \qquad ; \ell \in B \wedge [e]^\ell \in block_\star$$

در ادامه تابع gen را برای هر نوع بلوک تعریف می‌کنیم. توجه کنید که صرفاً در مواردی که تغییری در محیط انتساب نوع مربوط به گره صورت می‌گیرد آن را به‌صورت صریح بیان می‌کنیم، در بقیه موارد فرض می‌کنیم که محیط انتساب نوع منتسب به گره متناظر با بلوک یاد شده بدون تغییر باقی می‌ماند.

تابع انتقال برای $[\texttt{new } t]^\ell$: این دسته از عبارت‌ها رفتاری که از دیدگاه کشف رقابت داده اهمیت داشته باشد را از خود بروز نمی‌دهند. بنابراین، تابع انتقال برای آن‌ها نیز کار جالب توجهی انجام نمی‌دهد:

$$gen(\eta)([\texttt{new } t]^\ell) = \bot.$$

تابع انتقال برای $[var]^\ell$: این دسته از عبارت‌ها رفتاری که از دیدگاه کشف رقابت داده اهمیت داشته باشد را از خود بروز نمی‌دهند. بنابراین، تابع انتقال برای آن‌ها نیز کار جالب توجهی انجام نمی‌دهد:

$$gen(\eta)([var]^\ell) = \bot.$$

تابع انتقال برای $[\texttt{null}]^\ell$: این دسته از عبارت‌ها رفتاری که از دیدگاه کشف رقابت داده اهمیت داشته باشد را از خود بروز نمی‌دهند. بنابراین، تابع انتقال برای آن‌ها نیز کار جالب توجهی انجام نمی‌دهد:

$$gen(\eta)([\texttt{null}]^\ell) = \bot.$$

تابع انتقال برای $[(t)e]^\ell$: همان‌طور که پیش‌تر در ارائه نحو مجرد مشاهده کردیم، گراف جریان کنترل برای این عبارت طوری ساخته شده است که پیش از رسیدن به بلوک مربوط به این عبارت، به تمام رخدادهایی که توسط زیرعبارت e ممکن است تولید گردد رسیدگی شود. از آنجایی که تبدیل نوع یک عبارت رخدادی که برای کشف رقابت داده حائز اهمیت باشد را تولید نمی‌کند، تابع انتقال برای این دسته از بلوک‌ها نیز کار جالب توجهی انجام نمی‌دهد:

$$gen(\eta)([(t)e]^\ell) = \bot.$$

تابع انتقال برای $[e..fd]^\ell$: این دسته از عبارت‌ها یک رخداد از نوع خواندن را تولید می‌کنند:

$$gen(\eta)([e..fd]^\ell) = \begin{cases} (\{e\}, \mapsto_g) & ; \exists t \cdot P_\star; E_\star \vdash e. : \texttt{norace } t \\ \bot & ; o.w. \end{cases}$$

که در آن e و $\mapsto_g$ به‌صورت زیر تعریف می‌شود:

$$e = R(\ell, \Lambda^?, mayPointsTo(e.) \times \{fd\}),$$
$$\mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\dots, \Lambda^?, \dots) \wedge \eta = (E, \mapsto)\}.$$

تابع انتقال برای $[e_1.fd = e_2]^\ell$: این دسته از عبارت‌ها یک رخداد از نوع نوشتن را تولید می‌کنند:

$$gen(\eta)([e_1.fd = e_2]^\ell) = \begin{cases} (\{e\}, \mapsto_g) & ; \exists t \cdot P_\star; E_\star \vdash e_1 : \texttt{norace}\ t \\ \bot & ; o.w. \end{cases}$$

که در آن e و $\mapsto_g$ به‌صورت زیر تعریف می‌شود:

$$e = W(\ell, \Lambda^?, mayPointsTo(e_1) \times \{fd\}),$$
$$\mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\dots, \Lambda^?, \dots) \wedge \eta = (E, \mapsto)\}.$$

تابع انتقال برای $[begin(var = e_1; e_2)]^{\ell_n}$: این دسته از عبارت‌ها رفتاری که از دیدگاه کشف رقابت داده اهمیت داشته باشد را از خود بروز نمی‌دهند. بنابراین، تابع انتقال برای آن‌ها نیز کار جالب توجهی انجام نمی‌دهد:

$$gen(\eta)([begin(var = e_1; e_2)]^{\ell_n}) = \bot.$$

نقطه ورود به بلوک `let` جایی است که محیط انتساب نوع منتسب به گره متناظر با این دسته از بلوک‌ها تغییر می‌کند. در صورتی که $E_\star$ محیط انتساب نوع وراثتی گره مورد نظر باشد، محیط انتساب نوع منتسب به این گره چنین تعریف می‌شود:

$$E'_\star = E_\star \cup \{var : t \mid P_\star; E_\star \vdash e_1 : t\}$$

تابع انتقال برای $[end(var = e_1; e_2)]^{\ell_x}$: این دسته از عبارت‌ها رفتاری که از دیدگاه کشف رقابت داده اهمیت داشته باشد را از خود بروز نمی‌دهند. بنابراین، تابع انتقال برای آن‌ها نیز کار جالب توجهی انجام نمی‌دهد:

$$gen(\eta)([end(var = e_1; e_2)]^{\ell_x}) = \bot.$$

محیط انتساب نوع منتسب به گره متناظر با این دسته از بلوک‌ها نیز تغییر می‌کند. در صورتی که $E_\star$ محیط انتساب نوع وراثتی گره مورد نظر باشد، محیط انتساب نوع منتسب به این گره چنین تعریف می‌شود:

$$E'_\star = E_\star - \{var : t \mid P_\star; E_\star \vdash e_1 : t\}$$

تابع انتقال برای $[enterMonitor(e_1; e_2)]^{\ell_n}$: این دسته از عبارت‌ها یک رخداد از نوع ورود به مانیتور را تولید می‌کنند:

$$gen(\eta)([enterMonitor(e_1; e_2)]^{\ell_n}) = (\{e\}, \mapsto_g)$$

که در آن e و $\mapsto_g$ به‌صورت زیر تعریف می‌شود:

$$\begin{aligned}
&e = N(\ell_n, \Lambda^?, L),\\
&\mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\dots, \Lambda^?, \dots) \wedge \eta = (E, \mapsto)\},
\end{aligned}$$

به‌طوری که داریم:

$$L = \begin{cases} \emptyset & ; |mayPointsTo(e_1)| > 1 \\ & \quad \vee \exists n \in mayPointsTo(e_1) \cdot n = \mathtt{i}(\dots, *) \\ mayPointsTo(e_1) & ; o.w. \end{cases}$$

تابع انتقال برای $[exitMonitor(e_1; e_2)]^{\ell_x}$: این دسته از عبارت‌ها یک رخداد از نوع خروج از مانیتور را تولید می‌کنند:

$$gen(\eta)([exitMonitor(e_1; e_2)]^{\ell_x}) = (\{e\}, \mapsto_g)$$

که در آن e و $\mapsto_g$ به‌صورت زیر تعریف می‌شود:

$$\begin{aligned}
&e = X(\ell_x, \Lambda^?, L),\\
&\mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\dots, \Lambda^?, \dots) \wedge \eta = (E, \mapsto)\},
\end{aligned}$$

به‌طوری که داریم:

$$L = \begin{cases} \emptyset & ; |mayPointsTo(e_1)| > 1 \\ & \quad \vee \exists n \in mayPointsTo(e_1) \cdot n = \mathtt{i}(\dots, *) \\ mayPointsTo(e_1) & ; o.w. \end{cases}$$

تابع انتقال برای $[begin(var_1, \dots, var_k)]^{\ell_n}$: این دسته از عبارت‌ها رفتاری که از دیدگاه کشف رقابت داده اهمیت داشته باشد را از خود بروز نمی‌دهند. بنابراین، تابع انتقال برای آن‌ها نیز کار جالب توجهی انجام نمی‌دهد:

$$gen(\eta)([begin(var_1, \dots, var_k)]^{\ell_n}) = \bot.$$

تابع انتقال برای $[end(var_1, \ldots, var_k)]^{\ell_x}$: این دسته از عبارت‌ها رفتاری که از دیدگاه کشف رقابت داده اهمیت داشته باشد را از خود بروز نمی‌دهند. بنابراین، تابع انتقال برای آن‌ها نیز کار جالب توجهی انجام نمی‌دهد:

$$gen(\eta)([end(var_1, \ldots, var_k)]^{\ell_x}) = \bot.$$

تابع انتقال برای $[e.\texttt{start}()]^{\ell}$: در اینجا فرض می‌کنیم که خلاصه متد برای تمام متدهایی که ممکن است در این نقطه از برنامه به عنوان بدنه ریسه جدید فراخوانی شود موجود باشد. به‌طور دقیق‌تر فرض می‌کنیم:

$$\forall m \in callees(\ell) \cdot ms(m) \neq \emptyset,$$

با این حساب، تابع انتقال برای این دسته از عبارت‌ها در حالت کلی به‌صورت زیر تعریف می‌شود:

$$gen(\eta)([e.\texttt{start}()]^{\ell}) = (E_I, \mapsto_g)$$

به‌طوری که $(E_I, \mapsto_I) = inst\,(\ell, \bigsqcup\{s \mid ms(c'.\texttt{run}) = \{s\} \wedge c'.\texttt{run} \in callees(\ell)\})$ بوده، و داریم:

$$\mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\ldots, \Lambda^?, \ldots) \wedge e \in E_I \wedge \eta = (E, \mapsto)\} \cup \mapsto_I .$$

در تعریف بالا مجموعه‌های E_I و $\mapsto_I$ به‌صورت زیر تعریف می‌شوند.

$$(E_I, \mapsto_I) = i_3(\ell, i_2(\ell, i_1(\ell))),$$

همان‌طور که در فصل مربوط به تحلیل جریان کنترل و اشاره‌گر مشاهده خواهیم کرد، مفهوم گره‌های خارجی برای ساخت یک تحلیل اشاره‌گر پیمانه‌ای معرفی شده است. در تعریف بالا تابع i_1 مسئول نمونه‌سازی گره‌های خارجی است. در حالت کلی برای توسعه یک تحلیل اجرای مجرد می‌توان این تابع را نادیده گرفت؛ تابع i_1 صرفاً برای ارتباط این تحلیل با تحلیل اشاره‌گر ارائه شده در این پایان‌نامه تعریف شده است. برای تعریف این تابع، با فرض $M \subseteq N$، تابع کمکی map را به‌صورت زیر تعریف می‌کنیم:

$$map(M) = \bigcup\{g(n) \mid n \in M\}$$

که در آن، تابع g به‌صورت زیر تعریف می‌شود:

$$g(n) = \begin{cases} \{n\} & ; n \notin N_E \\ \{m \mid (n, m) \in \mu(\ell)\} & ; o.w. \end{cases}$$

که در آن تابع μ بخشی از خلاصه متد محاسبه شده توسط تحلیل اشاره‌گر برای متدهای فراخوانی شده در محل ایجاد ریسه

مورد نظر بوده، و N_E مجموعه گره‌های خارجی است. با توجه به تابع map می‌توان تابع $i_۱$ را به‌صورت زیر تعریف کرد:

$$i_۱(\ell,(E,\mapsto)) = (E',\mapsto')$$

تابع $i_۲$ برای نمونه‌سازی شناسه رخدادها به‌کار می‌رود. این تابع را به‌صورت زیر تعریف می‌کنیم:

$$i_۲(\ell,(E,\mapsto)) = (E',\mapsto')$$

که در آن E' و $\mapsto'$ به‌صورت زیر تعریف می‌شود:

$$\begin{aligned} E' &= E\left[\kappa(\ell s,\dots)/\kappa(s,\dots)\right], \\ \mapsto' &= \mapsto\left[\kappa(\ell s,\dots)/\kappa(s,\dots)\right]. \end{aligned}$$

تابع $i_۳$، مسئول نمونه‌سازی شناسه و چندگانگی ریسه‌ها برای آن دسته از رخدادهایی است که در محل مورد نظر ایجاد می‌شوند. برای تعریف این تابع ابتدا فرض می‌کنیم:

$$m = getLabMult(\ell)$$

حال تابع یاد شده را به‌صورت زیر تعریف می‌کنیم:

$$i_۳(\ell,(E,\mapsto)) = (E',\mapsto')$$

که در آن E' و $\mapsto'$ به‌صورت زیر تعریف می‌شود:

$$\begin{aligned} E' &= E\left[\kappa(\dots,\ell s^{m\otimes m'},\dots)/\kappa(\dots,s^{m'},\dots)\right], \\ \mapsto' &= \mapsto\left[\kappa(\dots,\ell s^{m\otimes m'},\dots)/\kappa(\dots,s^{m'},\dots)\right], \end{aligned}$$

توجه کنید که محدودیتی بر روی s نداریم.

توابع انتقال مربوط به فراخوانی متدها: آخرین بحث ما در مورد توابع انتقال برای عبارت‌های فراخوانی متدها است. شرط زیر باید برای اطمینان از وجود خلاصه برای آن دسته از متدهایی که در این مکان‌ها فراخوانی می‌شوند، و با متد فراخوانی کننده رابطه SCC ندارند برقرار باشد:

$$\forall m \in callees(\ell)\cdot(m,c.md)\notin SCC \implies ms(m)\neq\emptyset$$

حال دو تابع انتقال برای این دسته از عبارات را به‌صورت زیر تعریف می‌کنیم:

$$f^{call}_{\ell_c}(\eta) = \bot,$$
$$f^{ret}_{\ell_c,\ell_r}(\eta,\eta') = \eta \sqcup (E_I, \mapsto_g),$$

که در آن با فرض $inst'(\ell_c, \eta' \sqcup R) = (E_I, \mapsto_I)$ داریم:

$$\mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\dots, \Lambda^?, \dots) \wedge e \in E_I \wedge \eta = (E, \mapsto)\} \cup \mapsto_I,$$
$$R = \bigsqcup\{s \mid ms(m) = \{s\} \wedge m \in callees(\ell_c) \wedge (m, c.md) \notin SCC\}.$$

حال تابع نمونه‌سازی $inst'$ را با تعریف دقیق E_I و $\mapsto_I$ شناسایی می‌کنیم:

$$(E_I, \mapsto_I) = i'_۳(\ell_c, i_۲(\ell_c, i_۱(\eta')))$$

به‌طوری که توابع $i_۱$ و $i_۲$ مطابق قبل تعریف می‌شوند. تابع $i'_۳$ نیز مسئول نمونه‌سازی شناسه و چندگانگی ریسه‌ها برای آن دسته از محل‌های فراخوانی است که ممکن است ریسه‌ای در آن‌ها ایجاد شود. مطابق قبل، برای تعریف چنین تابعی ابتدا فرض می‌کنیم:

$$m = getLabMult(\ell)$$

حال تابع یاد شده را به‌صورت زیر تعریف می‌کنیم:

$$i'_۳(\ell, (E, \mapsto)) = (E', \mapsto')$$

که در آن E' و $\mapsto'$ به‌صورت زیر تعریف می‌شود:

$$E' = E\left[\kappa(\dots, \ell s^{m \otimes m'}, \dots)/\kappa(\dots, s^{m'}, \dots)\right],$$
$$\mapsto' = \mapsto\left[\kappa(\dots, \ell s^{m \otimes m'}, \dots)/\kappa(\dots, s^{m'}, \dots)\right],$$

به‌قسمی که $s \neq \Lambda$ باشد.

نکته. با توجه به توابع انتقال ارائه شده، مشاهده می‌شود که انتخاب شناسه یکتا در مکان‌های فراخوانی متدها و ایجاد ریسه‌ها باعث می‌شود که در رابطه $\mapsto$، در صورتی که آن را به‌صورت کمان‌های یک گراف تصور کنیم، هیچ سیکلی به وجود نیاید. به عبارتی بهتر، رابطه $\mapsto$ غیربازتابی است. از آنجایی که در حین افزودن یک رخداد به اجرای مجرد جاری تمام رخدادهای موجود در آن را به رخداد جدید متصل می‌کنیم، نتیجه می‌گیریم که رابطه یاد شده دارای خاصیت تعدی هست. از طرفی چون رخدادهای مربوط به ریسه‌های مختلف را به هم متصل نکرده‌ایم، و در نتیجه رابطه $\mapsto$ خالی از دور است، نتیجه می‌گیریم که است رابطه دارای خاصیت پادتقارنی نیز است. به این ترتیب رابطه $\mapsto$، همانند رابطه پیش‌رویدادی، یک ترتیب جزئی غیربازتابی بوده، و در حقیقت تخمینی ایمن از رابطه پیش‌رویدادی برای یک اجرای دلخواه $(E, \rightarrow)$ از برنامه

مورد نظر است که اجرای مجرد $(\widehat{E}, \mapsto)$ برای آخرین دستورالعمل متد اصلی آن برنامه منتسب شده است (خلاصه متد اصلی برنامه). برای چنین اجرای مجرد، و برای هر دو دستورالعمل i_1 و i_2 اثبات برآورده کردن شرط زیر همان اثبات درستی تحلیل همروندی است.

$$\forall e_1, e_2 \in \widehat{E} \cdot (e_1 \in \|i_1\| \wedge e_2 \in \|i_2\| \wedge e_1 \mapsto e_2) \\ \implies \left(\forall e_1', e_2' \in E \cdot e_1' \in [\![i_1]\!] \wedge e_2' \in [\![i_2]\!] \implies e_1' \to e_2'\right).$$

اما از آنجایی که در این پایان‌نامه تمرکز ما بر روی اثبات درستی سیستم مقابله با رقابت داده است، درستی سیستم کشف رقابت داده را ثابت می‌کنیم، که به‌صورت ضمنی می‌توان درستی تحلیل همروندی مورد استفاده در آن را نیز نتیجه گرفت.

۴.۳.۴ ایمنی اجراهای مجرد

همان‌طور که مشاهده کردیم اجراهای مجرد حاوی اطلاعات کاملی در مورد رخدادها و نیز ترتیب آن‌ها در هر ریسه است، در این بخش ضمن بررسی روش استخراج اطلاعات همروندی از روی اجراهای مجرد، مفهوم ایمنی اجراهای مجرد را بیان می‌کنیم. منظور ما از ایمنی، در این پایان‌نامه، نبود موارد رقابت داده در یک اجرای مجرد است.

برای بررسی همروندی دو رخداد کافی است بررسی کنیم که آیا مربوط به یک ریسه هستند، و یا اینکه آیا مربوط به ریسه‌های متفاوت هستند و هر یک توسط حداقل یک رخداد مانیتور محاصره شده‌اند. برای تعیین رخدادهای مانیتور که یک رخداد را محاصره کرده‌اند، در اولین مرحله، نیاز به تعیین رخدادهای ورود به مانیتور قبل از رخداد مورد نظر و رخدادهای خروج از مانیتور بعد از رخداد یاد شده داریم. برای این منظور ابتدا تعریفی از رخدادهای مانیتوری منطبق ارائه می‌دهیم.

تعریف ۵.۴. دو رخداد مانیتور را منطبق بر هم یا جفت هم گوییم هرگاه یکی از آن‌ها از نوع ورود و دیگری از نوع خروج از مانیتور بوده، و هر دو مربوط به یک شئ باشند. به عبارتی دقیق‌تر، رابطه دو موضعی $match(e_1, e_2)$ را برای دو رخداد $e_1 = \kappa_1(\dots, L_1)$ و $e_2 = \kappa_2(\dots, L_2)$ به‌صورت زیر تعریف می‌کنیم:

$$match(e_1, e_2) \iff \kappa_1 \neq \kappa_2 \wedge \kappa_1, \kappa_2 \in \{N, X\} \wedge L_1 = L_2.$$

■

با توجه به مفهوم اخذ و آزادسازی قفل‌ها، رخدادهای مانیتور منطبق، اثر همدیگر را خنثی می‌کنند. بنابراین، رخدادهای ورود و خروج از مانیتور قبل و بعد از یک رخداد (در هر طرف که باشند) یک ساختار پرانتزی می‌سازند، به‌طوری که تنها آن رخداد (های) ورود به مانیتور قبل از یک رخداد و رخداد (های) خروج از مانیتور بعد از آن که با هیچ رخداد مانیتور دیگر جفت (منطبق) نشده‌اند، رخداد مورد نظر را محاصره می‌کنند (و از اطلاعات دسترسی شده در آن محافظت می‌کنند.) رخدادهای ورود و خروج از مانیتور قبل از یک رخداد در یک اجرای مجرد η را به‌صورت زیر تعریف می‌کنیم:

$$N_\eta^{\leftarrow}(e) = \{e_N \mid e_N = N(\dots, \tau, \dots) \wedge e_N \mapsto e \wedge e_N, e \in E \wedge e = \kappa(\dots, \tau, \dots) \wedge \eta = (E, \mapsto)\},$$

$$X_\eta^{\leftarrow}(e) = \{e_X \mid e_X = X(\dots, \tau, \dots) \wedge e_X \mapsto e \wedge e_X, e \in E \wedge e = \kappa(\dots, \tau, \dots) \wedge \eta = (E, \mapsto)\}.$$

به‌صورت مشابه، رخدادهای ورود و خروج از مانیتور بعد از یک رخداد در یک اجرای مجرد η را به این صورت تعریف می‌کنیم:

$$N_\eta^{\rightarrow}(e) = \{e_N \mid e_N = N(\dots,\tau,\dots) \wedge e \mapsto e_N \wedge e, e_N \in E \wedge e = \kappa(\dots,\tau,\dots) \wedge \eta = (E,\mapsto)\},$$

$$X_\eta^{\rightarrow}(e) = \{e_X \mid e_X = X(\dots,\tau,\dots) \wedge e \mapsto e_X \wedge e, e_X \in E \wedge e = \kappa(\dots,\tau,\dots) \wedge \eta = (E,\mapsto)\}.$$

حال مجموعه $F(A,B)$ را به ازای دو مجموعه دلخواه از رخدادهای مانیتور اجرای مجرد η به‌صورت زیر تعریف می‌کنیم:

$$\begin{aligned} F(A,B) = \{f \mid f : A \to B \wedge & \text{ یک به یک } f \\ & \wedge \forall a \in A \cdot \forall b \in B \cdot f(a) = b \implies match(a,b) \\ & \wedge \forall a \in A \cdot \forall b \in B \cdot match(a,b) \implies [(\exists a' \in A \cdot f(a') = b) \vee (\exists b' \in B \cdot f(a) = b')]\}, \end{aligned}$$

همان‌طور که مشاهده می‌شود $F(A,B)$ شامل تمامی توابع یک به یک از A به B است که این توابع هر رخداد موجود در A را به رخداد قابل انطباق آن در B (در صورت وجود) نگاشت می‌کنند. واضح است که برای هر A و B غیرتهی، به‌طوری که حداقل یک $a \in A$ و یک $b \in B$ موجود باشد که $match(a,b)$، داریم: $F(A,B) \neq \emptyset$.

با توجه به مجموعه تعاریف صورت گرفته، می‌توان مجموعه تمام رخدادهای ورود به مانیتور قبل از یک رخداد را که با هیچ رخداد خروج از مانیتور قبل از آن رخداد جفت نشده است شناسایی کرد:

$$NetN_\eta^{\leftarrow}(e) = \bigcup\{N_\eta^{\leftarrow}(e) - ran(f) \mid f \in F(X_\eta^{\leftarrow}(e), N_\eta^{\leftarrow}(e))\}.$$

به‌طور مشابه، می‌توان مجموعه تمام رخدادهای خروج از مانیتور بعد از یک رخداد را که با هیچ رخداد ورود به مانیتور بعد از آن رخداد جفت نشده است شناسایی کرد:

$$NetX_\eta^{\leftarrow}(e) = \bigcup\{X_\eta^{\rightarrow}(e) - ran(f) \mid f \in F(N_\eta^{\rightarrow}(e), X_\eta^{\rightarrow}(e))\}.$$

واضح است که رخداد e توسط حداقل یک جفت رخداد ورود به مانیتور جفت نشده قبل، و خروج از مانیتور بعد از خود محاصره می‌گردد. در صورتی که حداقل یکی از این جفت رخدادها بر هم منطبق شوند، مکان حافظه دسترسی شده در رخداد مورد نظر توسط آن جفت محافظت می‌شود. در ادامه رابطه ord_η را برای بررسی ترتیب بین دو رخداد معرفی می‌کنیم. این رابطه برای دو رخداد از دو ریسه مختلف به‌طوری که دو رخداد توسط جفت رخدادهای مانیتور که بر هر منطبق هستند محاصره شده‌اند تعریف می‌شود.

در نهایت رابطه ord_η، برای بررسی وجود ترتیب علّی بین رخدادها (در تمام اجراهای ممکن)، را به‌صورت زیر تعریف می‌کنیم:

$$\begin{aligned} ord_\eta(e_1,e_2) \iff & ord'_\eta(e_1,e_2) \\ & \vee \exists (e_{N_1}, e_{X_1}, e_{N_2}, e_{X_2}) \in NetN_\eta^{\leftarrow}(e_1) \times NetX_\eta^{\rightarrow}(e_1) \times NetN_\eta^{\leftarrow}(e_2) \times NetX_\eta^{\rightarrow}(e_2) \cdot \\ & \quad \left(ord'_\eta(e_{N_1}, e_{X_2}) \wedge ord'_\eta(e_{N_2}, e_{X_1})\right), \end{aligned}$$

به‌طوری که $ord'(e_۱, e_۲)$ در آن، با فرض $\eta = (E, \mapsto)$، $e_۱, e_۲ \in E$، $e_۱ = \kappa_۱(id_۱, \tau_۱, L_۱)$، و $e_۲ = \kappa_۲(id_۲, \tau_۲, L_۲)$ به‌صورت زیر تعریف می‌شود:

$$ord'_\eta(e_۱, e_۲) \iff \tau_۱ = \tau_۲ \vee (\kappa_۱ \neq \kappa_۲ \wedge \kappa_۱, \kappa_۲ \in \{N, X\} \wedge L_۱ = L_۲ \wedge L_۱ \neq \emptyset).$$

نیازمندی عدم وجود رقابت داده: برای بیان نیازمندی عدم وجود رقابت داده در یک اجرای مجرد، نیاز داریم تا رخدادهای متضاد و رخدادهای نوشتن چندگانه را که در یک اجرای مجرد شناسایی کنیم. به این صورت که اگر اجرای مجری حاوی این نوع رخدادها باشد، به عنوان یک اجرای مجرد حاوی رقابت داده شناسایی می‌شود.

تعریف ۶.۴. رخدادهای $e_۱ = \kappa_۱(id_۱, \tau_۱, L_۱)$ و $e_۲ = \kappa_۲(id_۲, \tau_۲, L_۲)$ را با فرض $e_۱, e_۲ \in E$ و $\eta = (E, \mapsto)$ را متضاد گویند هرگاه توسط ریسه‌های مختلف صورت گیرند، حداقل یکی از آن‌ها از نوع نوشتن بوده، و هر دو رخداد به حداقل یک نقطه مشترک حافظه دسترسی داشته باشند. این موضوع را با $conf(e_۱, e_۲)$ نشان داده، و به‌صورت زیر تعریف می‌کنیم:

$$conf(e_۱, e_۲) \iff \kappa_۱ = W \wedge (\kappa_۲ = W \vee \kappa_۲ = R) \wedge \tau_۱ \neq \tau_۲ \wedge L_۱ \cap L_۲ \neq \emptyset.$$

■

تعریف ۷.۴. رخداد $e = \kappa(id, \tau^m, L)$ را نوشتن چندگانه گوییم، هرگاه از نوع نوشتن باشد و چندگانگی ریسه تولید کننده آن $*$ باشد. این موضوع را با $mwr(e)$ نشان داده، و به‌صورت زیر تعریف می‌کنیم:

$$mwr(e) \iff \kappa = W \wedge m = *.$$

■

در نهایت نیازمندی عدم وجود رقابت داده در یک اجرای مجرد $\eta = (E, \mapsto)$ به‌صورت زیر تعریف می‌کنیم:

$$NORACE(\eta) \iff (\forall e \in E \cdot \neg mwr(e)) \wedge (\forall e_۱, e_۲ \in E \cdot conf(e_۱, e_۲) \implies ord(e_۱, e_۲)).$$

در انتهای این فصل برای تحلیل بین‌رویه‌ای ثابت می‌کنیم که اگر گزاره یاد شده برای اجرای مجرد منتسب به گره پایانی عبارت آغازین یک برنامه برقرار باشد، هر اجرای آن برنامه خالی از خطای رقابت داده است.

۵.۳.۴ تحلیل پیمانه‌ای

همان‌طور که پیش‌تر نیز اشاره کردیم، تحلیل ارائه شده در بخش ۳.۳.۴ یک تحلیل پیمانه‌ای واقعی نیست، چرا که واقعاً هر متد را در انزوا تحلیل نمی‌کند بلکه متدهایی که به‌صورت بازگشتی متقابل هستند را با هم در نظر گرفته و معادلات جریان داده را برای کل آن‌ها تشکیل و حل می‌کند. همچنین، به علت رشد بدون محدودیت شناسه رخدادها در مکان‌های فراخوانی بازگشتی متدها الگوریتمی (رویه‌ای که در زمان متناهی خاتمه پیدا کند) برای محاسبه پاسخ معادلات تشکیل شده وجود ندارد. از این رو نیاز داریم تحلیلی دیگر را که محاسبه‌پذیر و نیز به‌صورت کامل پیمانه‌ای است ارائه دهیم. علت ارائه تحلیل

بین‌رویه‌ای نیز در این است که تحلیل بین‌رویه‌ای بسیار ساده‌تر از تحلیل پیمانه‌ای اثبات می‌شود.

تمامی بخش‌های این تحلیل به جز تابع انتقال برای فراخوانی متدها با تحلیل بین‌رویه‌ای مشترک است. در این تحلیل برای جلوگیری از رشد نامحدود شناسه برچسب‌ها در مکان‌های فراخونی بازگشتی متدها از تحلیل بازگشتی متدها خودداری می‌کنیم. منظور ما از اجتناب از تحلیل بازگشتی این است که اگر در روند محاسبه پاسخ تحلیل برای یک متد نیاز به محاسبه جواب همان متد داشتیم، به جای محاسبه مجدد جواب از مقدار $\bot$ استفاده کنیم. ما این کار را با افزودن یک پارمتر بیشتر به معادلات جریان داده انجام داده‌ایم، این پارامتر در اصل یک پشته فراخوانی بوده، و cs نام دارد. ماهیت این پارامتر یک مجموعه از اسامی متدها است که نشان دهنده مجموعه‌ای از متدهایی است که قرار است پاسخ تحلیل را برای آن‌ها محاسبه کنیم. در صورتی که در حین یافتن جواب تحلیل برای هر یک از این متدها نیاز به تحلیل متدی پیدا پیدا کردیم، نام متد مورد نظر را پشته فراخوانی قرار داده و ضمن رها کردن محاسبه پاسخ برای تحلیل جاری، به محاسبه پاسخ متد فراخوانی شده می‌پردازیم. در صورتی که نام متد فراخوانی شده پیش‌تر در پشته موجود باشد، به جای محاسبه پاسخ تحلیل برای متد مورد نظر از مقدار $\bot$ به جای جواب تحلیل استفاده می‌کنیم.

در ادامه این بخش ضمن ارائه تحلیلی مبتنی بر جریان داده که بر اساس این ایده طراحی شده است، الگوریتمی برای تبیین ترتیب تحلیل متدهای یک برنامه به منظور برقرار شدن پیش‌شرط‌های مورد انتظار تحلیل ارائه می‌شود. درستی این الگوریتم را اثبات کرده‌ایم. بنابراین، تضمین می‌کنیم که تحلیل پیمانه‌ای برای هر برنامه جاوای همروند قابل استفاده است. در انتهای این بخش ثابت می‌کنیم که استفاده از مقدار $\bot$ به‌جای محاسبه مجدد جواب برای یک تحلیل تأثیری در میزان رسایی اطلاعات محاسبه شده ندارد. به عبارتی دیگر، اطلاعات محاسبه شده توسط تحلیل پیمانه‌ای، از دیدگاه کشف رقابت داده، معادل با اطلاعات بدست آمده از تحلیل بین‌رویه‌ای است. بنابراین، اثبات درستی برای تحلیل بین‌رویه‌ای درستی تحلیل پیمانه‌ای را نیز در پی دارد.

تحلیل پیمانه‌ای ارائه شده در این بخش مبتنی بر جریان داده، رو به جلو، و با قطعیت may است. همانند قبل بدون از دست دادن کلیت مسأله تحلیل را برای یک متد $c.md$ از برنامه $P_\star$ ارائه می‌دهیم. گراف جریان کنترل مورد استفاده گراف جریان کنترل پیمانه‌ای متد یاد شده، یا همان $MCFG^{P_\star}_{c.md} = (B, F)$، است. تحلیل ارائه شده یک نمونه از چارچوب یکنوا است، و می‌توان آن را با استفاده از یک شش‌تایی به‌صورت زیر بیان کرد:

$$\mathsf{AEA}^{\mathsf{mod}} = (\mathcal{AE}, \mathcal{F}, F, E, \iota, f_\cdot),$$

که در آن تمام مؤلفه‌های تحلیل همانند تحلیل بین‌رویه‌ای تعریف می‌شود، به جز مجموعه $\mathcal{F}$ که آن را به‌صورت زیر تعریف می‌کنیم.

$$\mathcal{F} = \{f : AE \to AE \mid f \text{ یکنوا}\}.$$

در ارائه این تحلیل نیز به اطلاعات نوع زیرعبارت‌های بدنه متدها نیاز داریم. شیوه انتساب محیط‌های انتساب نوع به گره‌های گراف جریان کنترل، و به‌روزرسانی آن‌ها مطابق قبل انجام می‌شود.

دستگاه معادلات جریان داده که در آن $AEA^{\mathsf{mod}}_\circ(\ell)$ و $EAE^{\mathsf{mod}}_\bullet(\ell)$ به ازای $\ell \in Labels_\star$ متغیرهای مجهول دستگاه هستند. دستگاه یاد شده را برای این تحلیل را به‌صورت زیر تعریف می‌کنیم:

$$AEA^{\mathsf{mod}}_\circ(\ell) = \bigsqcup\{AEA^{\mathsf{mod}}_\bullet(\ell') \mid (\ell', \ell) \in F\} \qquad ; \ell \in B \tag{۵.۴}$$

همچنین، برای هر برچسب ℓ در مجموعه B (اعم از آن‌هایی که برای برچسب‌گذاری مکان‌های فراخوانی متدها به‌کار رفته‌اند) داریم:

$$AEA^{\mathsf{mod}}_{\bullet}(\ell) = f_\ell(cs)(AEA^{\mathsf{mod}}_{\circ}(\ell)) \tag{۶.۴}$$

تعریف ۸.۴ (پاسخ تحلیل پیمانه‌ای). زوج تابع $(AEA^{\mathsf{mod}}_{\circ}, AEA^{\mathsf{mod}}_{\bullet})$ که در معادلات (۵.۴) و (۶.۴) برای یک متد $c.md$ در برنامه $P_\star$، یک پشته فراخوانی cs حاوی $c.md$، و خلاصه متد ms صدق می‌کند، پاسخ تحلیل اجرای مجرد پیمانه‌ای برای متد یاد شده نام دارد. این پاسخ را در قالب یک تابع به‌صورت زیر بیان می‌کنیم:

$$ModAnalyse(P_\star, c.md, cs, ms) = \{(\circ, AEA^{\mathsf{mod}}_{\circ}), (\bullet, AEA^{\mathsf{mod}}_{\bullet})\} \qquad ; c.md \in cs$$

■

تنها تغییر در تعریف خلاصه متدها این است که، از این به بعد، به جای استفاده از تابع $IntAnalyse$ برای بدست آوردن خلاصه متد از $ModAnalyse$ استفاده می‌کنیم. حال با تعریف توابع انتقال، تحلیل پیمانه‌ای را به‌طور کامل ارائه می‌دهیم.

صورت کلی توابع انتقال به فرم زیر تعریف می‌شود:

$$f_\ell(cs)(\eta) = \eta \sqcup gen'(cs, \eta)([B]^\ell) \qquad ; [B]^\ell \in blocks_\star$$

به‌طوری که gen' برحسب تابع gen، که پیش‌تر در بخش ۳.۳.۴ معرفی کردیم، قابل بیان است:

$$gen'(cs, \eta)([B]^\ell) = \begin{cases} gen''(cs, \eta)([B]^\ell) & ; \ell \text{ برچسب یک مکان فراخوانی است} \\ gen(\eta)([B]^\ell) & ; o.w. \end{cases}$$

در ادامه با تعریف تابع gen'' برای مکان‌های فراخوانی ریسه‌ها تعریف تحلیل را تکمیل می‌کنیم.

تابع انتقال برای $[e..md(e_1, \ldots, e_k)]^\ell$: در تعریف تابع انتقال برای این دسته از عبارت‌ها فرض می‌کنیم که خلاصه متد برای تمام متدهایی که ممکن است در این نقطه از برنامه فراخوانی شود، و هیچ‌کدام از متدهای موجود در cs را به‌صورت بازگشتی فراخوانی نکند، موجود باشد. به‌طور دقیق‌تر فرض می‌کنیم:

$$\forall m \in callees(\ell) \cdot (m \notin cs \wedge \forall m' \in cs \cdot (m, m') \notin SCC) \implies ms(m) \neq \emptyset \tag{۷.۴}$$

همچنین، فرض می‌کنیم بدنه تمام متدهایی که در نقطه یاد شده فراخوانی شود و حداقل یکی از متدهای موجود در cs را به‌صورت بازگشتی فراخوانی کند، موجود باشد. به عبارتی بهتر، فرض می‌کنیم:

$$\forall m \in callees(\ell) \cdot (m \notin cs \wedge \exists m' \in cs \cdot (m, m') \in SCC) \implies bodyOf(m) \neq \Lambda \tag{۸.۴}$$

حال تابع انتقال را به‌صورت زیر تعریف می‌کنیم:

$$gen''(cs,a)([e..md(e_1,\ldots,e_k)]^\ell) =$$
$$inst'(\bigsqcup\{s \mid ms(m)=\{s\} \wedge m \in callees(\ell) \wedge m \notin cs \wedge \forall m' \in cs \cdot (m,m') \notin SCC\}$$
$$\sqcup(\bigsqcup\{ModAnalyse(P_\star, c.md, cs \cup \{m\}, ms)(\bullet)(final(bodyOf(m))) \mid$$
$$m \in callees(\ell) \wedge m \notin cs \wedge (\exists m' \in cs \cdot (m,m') \in SCC)\}))$$

که در آن تابع $inst'$ مطابق قبل تعریف می‌شود. همان‌طور که مشاهده می‌شود مقدار محاسبه شده برای متدهایی که پشته فراخوانی قرار دارند، برابر با $\bot$ است. پیش‌شرط (۷.۴) این حس را به‌وجود می‌آورد که ممکن است، در تحلیل یک برنامه، حالتی پیش آید که در آن شرط یاد شده برقرار نباشد. در ادامه الگوریتمی ارائه می‌دهیم که ترتیبی برای تحلیل متدهای یک برنامه دلخواه ارائه می‌دهد که این ترتیب تضمین می‌کند که تمامی متدهای برنامه مورد نظر در نهایت تحلیل شده و در هر یک از این تحلیل‌ها شرط (۷.۴) باشد.

۶.۳.۴ ترتیب تحلیل متدها

در این بخش الگوریتمی برای ارائه یک ترتیب برای تحلیل متدهای یک برنامه دلخواه (با استفاده از تحلیل پیمانه‌ای بخش ۵.۳.۴) معرفی می‌شود. این ترتیب تضمین می‌کند که (۱) پیش‌شرط ۷.۴ در هر بار فراخوانی تحلیل برقرار باشد، (۲) تمامی متدهای برنامه مورد نظر در نهایت تحلیل شوند. در ادامه ضمن ارائه الگوریتم، درستی آن را نیز اثبات کرده‌ایم.

قبل ارائه الگوریتم لازم است که چند تعریف مربوط به توابعی که به‌صورت بازگشتی متقابل تعریف شده‌اند ارائه دهیم. در صورتی که m عضو $Methods_\star$ یک متد از برنامه دلخواه $P_\star$ باشد، **گروه بازگشتی** متد m را با $recGroup(m)$ نشان داده، و به عنوان مجموعه تمام متدهایی که با m به‌صورت بازگشتی متقابل تعریف شده‌اند در نظر می‌گیریم. این مجموعه را به‌صورت زیر تعریف می‌کنیم:

$$recGroup(m) = \{m' \mid m' \in Methods_\star \wedge (m,m') \in SCC\},$$

با توجه به گروه بازگشتی یک متد، می‌توان گروه بازگشتی برای مجموعه‌ای متدها $M \subseteq Methods_\star$ تعریف کرد:

$$RecGroups(M) = \bigcup\{recGroup(m) \mid m \in M\}.$$

با توجه به تعریف‌های صورت گرفته، الگوریتم یاد شده را در شکل ۵.۴ ارائه می‌دهیم. توجه کنید که در این الگوریتم از زیرروال $\textsc{DoAnalyse}(m)$ برای خلاصه کردن عملیات: (۱) تحلیل متد m با استفاده از تحلیل پیمانه‌ای بخش ۵.۳.۴، و (۲) تنظیم خلاصه متدها، به‌طور که تابع ms متد m را به مقدار محاسبه شده در گام (۱) نگاشت کند.

قضیه ۱.۴. شرایط زیر برای الگوریتم ۵.۴ بر قرار است:

- پیش‌شرط ۷.۴ در هر فراخوانی زیرروال *DoAnalyse* بر قرار است،
- الگوریتم خاتمه می‌یابد.

الگوریتم: تعیین ترتیب تحلیل متدهای یک برنامه

ورودی: برنامه $P_\star$ و مجموعه $Methods_\star$

خروجی: حالت سیستم کشف رقابت داده که در آن تمامی متدها تحلیل شده‌اند

```
1: M = ∅;
2: N = Methods⋆;
3: do
       for each m ∈ N do (۱)
           if mayCall(m) ⊆ M then (⋆)
               DoAnalyse(m);
               M = M ∪ {m};
       for each r ∈ RecGroups(N) do (۲)
           for each m ∈ r do
               if mayCall(m) ⊆ (r ∪ M) then (⋆⋆)
                   DoAnalyse(m);
                   M = M ∪ {m};
       N = N − M;
4:
5: while N ≠ ∅;
```

شکل ۵.۴: الگوریتم تعیین ترتیب تحلیل متدهای یک برنامه

اثبات. با توجه به شرط‌های (⋆) و (⋆⋆) واضح است که هنگامی که زیرروال DoAnalyse برای تحلیل متد m مورد استفاده قرار می‌گیرد، تمامی متدهایی که ممکن است توسط این متد فراخوانی می‌شوند، پیش‌تر تحلیل شده و بنابراین خلاصه برای آن موجود، یا اینکه با متد مورد نظر در یک گروه بازگشتی قرار دارند، که در این صورت نیازی به خلاصه متد وجود ندارد.

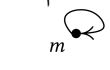
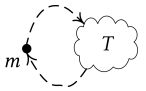
حال باید ثابت کنیم که حلقه **do** − **while** خطوط ۳ تا ۵ برای هر مجموعه $Methods_\star$ خاتمه می‌یابد. برای این کار بر روی اندازه این مجموعه استقرا می‌زنیم:

$|Methods_\star| = ۱$: در این حالت گراف فراخوانی برنامه مشابه موارد (۱) یا (۲) در جدول ۲.۴ است. حالت (۱) با توجه به این که $mayCall(m)$ برابر با تهی است، توسط حلقه (۱) اداره می‌شود. حالت (۲) نیز با توجه به این که $r = \{m\}$ تنها گروه بازگشتی برای این مجموعه از متدها است، داریم $mayCall(m) \subseteq r$ و در نتیجه توسط حلقه (۲) اداره می‌شود. در هر دو حالت با اضافه شدن m به مجموعه M الگوریتم خاتمه می‌یابد.

فرض استقرا: الگوریتم برای مجموعه $Methods_\star$ با اندازه کمتر از k، به‌طوری که $k > ۱$ است، خاتمه می‌یابد.

$|Methods_\star| = k$: حال برای تکمیل اثبات قضیه کافی است تمام حالاتی که گراف فراخوانی برای این مجموعه از متدها می‌تواند داشته باشد بررسی می‌کنیم. این حالات در جدول ۲.۴، در خانه‌های (۳) تا (۱۰)، جدول فهرست شده‌اند، که در آن

جدول ۲.۴: حالت‌های مختلف شکل یک گراف فراخوانی هنگامی که متد m را به آن اضافه می‌کنیم. جزئی که با یک ابر نشان داده شده است، زیرگرافی از گراف فراخوانی است که حاوی گره m نیست. کمان با خط ممتد نشان دهنده فراخوانی بازگشتی متد m توسط خود آن متد است، و کمان‌ها با خطچین نشان دهنده یک یا چند رابطه فراخوانی بین متد m و متد(های) موجود در زیرگراف T است. جهت کمان نشان می‌دهد که کدام متد کدام متد(ها) را فراخوانی می‌کند.

	(۲)		(۱)
	(۴)		(۳)
	(۶)		(۵)
	(۸)		(۷)
	(۱۰)		(۹)

ابر نشان دهنده زیرگرافی از گراف فراخوانی که دارای تعداد گره‌های کمتر از k است. در هر مورد (۳) و (۴) زیرگراف T با توجه به فرض استقرا تحلیل شده، و متد m در (۳) توسط حلقه (۱) و در (۴) توسط حلقه (۲) اداره شده، و با افزودن متد m به M الگوریتم خاتمه می‌یابد.

در حالت‌ها (۵) و (۶) تحلیل m به ترتیب با استفاده از حلقه‌های (۱) و (۲) اداره می‌شود، و مقدار M و N به‌ترتیب برابر با $\{m\}$ و $Methods_\star - \{m\}$ انتخاب می‌شود. حال الگوریتم با یک مجموعه متد با اندازه کمتر از k مواجه است که با توجه به فرض استقرا می‌توان گفت که الگوریتم برای این ورودی خاتمه می‌یابد.

در هر دو حالت (۷) و (۸) الگوریتم برای زیرگراف T خاتمه می‌یابد، تحلیل m به ترتیب توسط حلقه‌های (۱) و (۲) اداره (به M اضافه کرده و از N حذف می‌کند) می‌شود. بنابراین، الگوریتم در این دو حالت نیز خاتمه پیدا می‌کند.

در نهایت تنها موارد بررسی نشده، حالت‌های (۹) و (۱۰) است. حالت (۹) را بررسی می‌کنیم، بررسی حالت (۱۰) به‌صورت مشابه انجام می‌شود. هنگامی که متد m با تمام متدهایی از T که فراخوانی می‌کند (یا فراخوانی می‌شود) رابطه SCC دارد، تمام این متدها (به همراه خود m) در یک گروه بازگشتی قرار می‌گیرند. بنابراین، مجموعه گره‌های زیرگراف T به دو زیرمجموعه T_{SCC} و T'_{SCC} افراز می‌شود، که اولی مجموعه تمام متدهایی را نشان می‌دهد که با m در یک گره بازگشتی هستند (این مجموعه شامل خود m نیز هست)، و دومی حاوی تمامی متدهایی است که با m در یک گروه بازگشتی نیستند. واضح است که $|T'_{SCC}| < k$ بوده، و بنابر فرض استقرا، الگوریتم برای این زیرمجموعه خاتمه می‌یابد. مجموعه متدهای T_{SCC} نیز با $|T_{SCC}| \leq k$ با استفاده از حلقه دوم اداره شده و الگوریتم خاتمه می‌یابد. در حالتی هم که m با هیچ کدام از متدهای موجود در T رابطه SCC ندارد، اثبات قضیه به راحتی و با توجه به موارد (۵) و (۷) در جدول ۲.۴ قابل انجام است. تنها حالتی که بررسی نکردیم، حالتی است که در آن $T_{SCC} \neq \emptyset$ و نیز متدهایی از T متد m را فراخوانی می‌کنند که با آن

رابطه SCC ندارند. این مجموعه را با T_m نشان داده، و واضح است که $T_m \neq \emptyset$ است. همچنین متد m متدهایی از T را فراخوانی می‌کند که با آن‌ها رابطه SCC ندارد. این مجموعه را با m_T نشان داده، و می‌دانیم $m_T \neq \emptyset$ است. واضح است که $|m_T| < k$ بوده، و با توجه به این که متدهای موجود در این مجموعه با m در یک گروه بازگشتی نیستند، تحلیل آن‌ها وابسته به تحلیل m یا هر متدی که با آن در یک گروه بازگشتی قرار دارد نیست. بنابراین، با توجه به فرض استقرا الگوریتم برای این زیرمجموعه از $Methods_\star$ خاتمه می‌یابد. حال داریم:

$$M = m_T,$$
$$N = Methods_\star - m_T,$$

با توجه به این‌که m_T غیر تهی است، الگوریتم با یک مجموعه متد با اندازه کمتر از k مواجه می‌شود، که طبق فرض استقرا الگوریتم به ازای این ورودی نیز خاتمه می‌یابد. ■

مطابق این قضیه الگوریتم برای هر مجموعه متد دلخواه (و در نتیجه هر برنامه دلخواه $P_\star$) خاتمه می‌یابد. این یعنی تمام متدهای موجود در برنامه مورد نظر ضمن برقرار بودن پیش‌شرط ۷.۴ تحلیل می‌شوند. در ادامه ثابت می‌کنیم که پاسخ دو تحلیل پیمانه‌ای و بین‌رویه‌ای از دیدگاه کشف رقابت داده با هم معادل هستند. بنابراین، اثبات درستی تحلیل بین‌رویه‌ای درستی تحلیل پیمانه‌ای را در بر دارد. درنتیجه می‌توان ابزاری (محاسبه‌پذیر) درست برای کشف رقابت داده ارائه داد.

۷.۳.۴ هم‌ارزی دو تحلیل بین‌رویه‌ای و پیمانه‌ای

با توجه به اینکه توابع انتقال برای دو تحلیل بین‌رویه‌ای و پیمانه‌ای همگی یکنوا هستند، طبق قضیه نقطه ثابت تارسکی (قضیه ۳.۲ در فصل ۲) پاسخی برای دستگاه معادلات جریان داده تشکیل شده در دو تحلیل موجود است. در این بخش ثابت می‌کنیم که این دو جواب از دیدگاه کشف رقابت داده با هم معادل هستند. نتیجه مهمی که از این معادل بودن به دست می‌آوریم این است که می‌توان درستی تحلیل پیمانه‌ای را با توجه به درستی تحلیل بین‌رویه‌ای نتیجه گرفت. در ادامه این بخش، ابتدا قضیه‌ای برای نشان دادن این که رخدادهای تولید شده توسط تحلیل پیمانه‌ای توسط تحلیل بین‌رویه‌ای نیز تولید می‌شود، ارائه می‌دهیم. در نهایت نشان می‌دهیم که جواب محاسبه شده برای یک متد توسط تحلیل پیمانه‌ای به ازای برچسب نهایی آن متد ایمن است اگر و تنها اگر همین مقدار برای پاسخ بدست آمده از طریق تحلیل بین‌رویه‌ای ایمن باشد (لازم به یاد آوری است که اجرای مجرد η را ایمن گوییم هرگاه بتوانیم نشان دهیم $NORACE(\eta)$).

لم ۲.۴. متد $c.md$ از برنامه $P_\star$ مفروض است. فرض کنید که برای هر متد m در $mayCall(c.md)$ که با $c.md$ در یک گروه بازگشتی نیست، اگر η^{mod}_m نشان دهنده مقدار جواب تحلیل پیمانه‌ای، و η^{int}_m نشان دهنده مقدار جواب تحلیل بین‌رویه‌ای به ازای گره پایانی بدنه m باشد، داشته باشیم: $\eta^{\mathsf{mod}}_m \sqsubseteq \eta^{\mathsf{int}}_m$. آنگاه با فرض $\eta^{\mathsf{mod}}_{c.md}$ به عنوان مقدار جواب تحلیل پیمانه‌ای، و $\eta^{\mathsf{int}}_{c.md}$ به عنوان مقدار جواب تحلیل بین‌رویه‌ای به ازای گره پایانی بدنه $c.md$، داریم: $\eta^{\mathsf{mod}}_{c.md} \sqsubseteq \eta^{\mathsf{int}}_{c.md}$.

اثبات. با توجه به تعریف گراف جریان کنترل برای دو نوع تحلیل، مجموعه گره‌های گراف‌های جریان کنترل برای متد $c.md$ در دو تحلیل بین‌رویه‌ای و پیمانه‌ای حاوی برچسب‌هایی است که در بدنه متد یاد شده ظاهر می‌شوند. همچنین، گراف جریان کنترل پیمانه‌ای زیرگرافی از گراف جریان کنترل بین‌رویه است، به‌طوری که گراف جریان کنترل پیمانه‌ای فاقد جریان‌های بین‌رویه‌ای، و گره‌هایی به غیر از برچسب‌های ظاهر شده در بدنه $c.md$ است. بنابراین، برای یک گراف جریان کنترل پیمانه‌ای

(B, F) برای متد یاد شده داریم:

$$\bigsqcup\{AEA^{\mathsf{mod}}_{\bullet} \mid (\ell', \ell) \in F\} = \bigsqcup\{AEA^{\mathsf{mod}}_{\bullet} \mid (\ell', \ell) \in F \vee (\ell'; \ell) \in F\}.$$

از طرفی مقدار اولیه انتخاب شده برای هر دو تحلیل، و نیز تمامی توابع انتقال به جز تابع انتقال برای عبارت‌های فراخوانی متدها با هم برابر است. حال برای تکمیل اثبات قضیه کافی است ثابت کنیم که تمام رخدادهای تولید شده توسط تابع انتقال فراخوانی متدها در تحلیل پیمانه‌ای توسط همان تابع از تحلیل بین‌رویه‌ای نیز تولید می‌شود. تابع انتقال برای عبارت‌های فراخوانی متدها در تحلیل پیمانه‌ای، به ازای تمام متدهای فراخوانی شده در یک مکان فراخوانی که با $c.md$ در یک گروه بازگشتی قرار دارند از خلاصه آن متدها برای تولید رخدادهای ایجاد شده در اثر فراخوانی آن متدها استفاده می‌کند. با توجه به فرض قضیه و تعریف خلاصه متدها به راحتی می‌توان مشاهده کرد که اگر ms^{mod} خلاصه متدها برای تحلیل پیمانه‌ای و ms^{int} خلاصه متدها برای تحلیل بین‌رویه‌ای باشد، داریم: $ms^{\mathsf{mod}}(m) \sqsubseteq ms^{\mathsf{int}}(m)$. از طرفی از آنجایی که تابع یاد شده برای متدهایی که با $c.md$ در یک گروه بازگشتی هستند و در cs موجود هستند هیچ رخدادی تولید نمی‌کند. بنابراین، هر رخداد تولید شده توسط توابع انتقال تحلیل پیمانه‌ای توسط توابع انتقال تحلیل بین‌رویه‌ای نیز تولید می‌شود. ■

نشان دادیم که اگر $\eta_m = (E_m, \mapsto_m)$ خلاصه متد محاسبه شده توسط تحلیل پیمانه‌ای و $\eta_i = (E_i, \mapsto_i)$ خلاصه متد محاسبه شده توسط تحلیل بین‌رویه‌ای برای متد $c.md$ باشد، داریم: $\eta_m \sqsubseteq \eta_i$، یا به عبارتی $E_m \subseteq E_i$ و $\mapsto_m \subseteq \mapsto_i$. در ادامه ثابت می‌کنیم که خلاصه متد محاسبه شده توسط دو تحلیل از دیدگاه کشف رقابت داده با هم معادل هستند. قبل از ارائه لم مربوط به این ادعا، لازم است به نکته‌ای مربوط به طبیعت دو تحلیل اشاره کنیم. در صورتی که فرض کنیم $E = E_i - E_m$ مجموعه تمام رخدادهایی باشد که توسط تحلیل پیمانه‌ای تولید نمی‌شود، داریم:

$$\forall e \in E \cdot \exists e' \in E_m \cdot \exists s \in L^+_{call} \cdot getEID(e) = s.getEID(e'), \tag{۹.۴}$$

و

$$\forall e \in E \cdot \exists e' \in E_m \cdot \exists s \in L^+_{call} \cdot getTID(e) = s.getTID(e'), \tag{۱۰.۴}$$

که در این معادلات توابع $getEID$ و $getTID$ به ترتیب برای به‌دست آوردن شناسه یک رخداد $e \in Events$ و شناسه ریسه آن رخداد به‌کار می‌رود. این توابع به‌صورت زیر قابل تعریف هستند:

$$\begin{aligned} e = \kappa(id, \tau, L) : \\ getEID(e) = id, \\ getTID(e) = \tau. \end{aligned}$$

همچنین، مجموعه L_{call} به عنوان تمام برچسب‌های موجود در متد $c.md$ و تمام متدهایی که به‌طور مستقیم و غیرمستقیم

توسط این متد فراخوانی شده و یک مکان فراخوانی متد را برچسب‌گذاری می‌کند تعریف می‌کنیم:

$$L_{call} = \{\ell \mid \ell \in labelsOf(c.md) \wedge \text{است } \ell \text{ بر چسب یک مکان فراخوانی}\}$$

که در آن تابع $labelsOf$ مطابق تعریف ارائه شده در ۱.۴ فرض شده است.

لم ۳.۴. اگر $\eta_m = (E_m, \mapsto_m)$ خلاصه متد محاسبه شده توسط تحلیل پیمانه‌ای و $\eta_i = (E_i, \mapsto_i)$ خلاصه متد محاسبه شده توسط تحلیل بین‌رویه‌ای برای متد $c.md$ باشد، همان‌طور که مشاهده کردیم، $\eta_m \sqsubseteq \eta_i$ است. با توجه به این فرض داریم:

$$NORACE(\eta_m) \iff NORACE(\eta_i).$$

اثبات. با توجه به فرض قضیه داریم:

$$E_m \subseteq E_i,$$
$$\mapsto_m \subseteq \mapsto_i .$$

بنابراین، به راحتی می‌توان مشاهده کرد که طرف برگشت قضیه ($NORACE(\eta_i) \implies NORACE(\eta_m)$) برقرار است. در ادامه به اثبات طرف رفت قضیه می‌پردازیم.

از آنجایی که تابع $mayPointsTo$ مورد استفاده در دو تحلیل برابر فرض شده، و نیز فرض کرده‌ایم که تحلیل اشاره‌گر مورد استفاده برای تعریف این تابع درست است، نتیجه می‌گیریم که مؤلفه سوم رخدادهای تولید شده توسط دو تحلیل (مکان حافظه یا شئ مورد دسترسی حین وقوع رخداد) با هم برابر است. از طرفی با توجه تعریف چندگانگی برچسب‌ها در بخش ۴.۲.۴، و نیز با توجه به توابع انتقال مربوط به دسترسی به محتوای ارجاعات و انتساب، که در دو تحلیل با هم برابر است، نتیجه می‌گیریم که چندگانگی و نوع رخدادهای تولید شده توسط دو تحلیل با هم برابر است. بنابراین داریم:

$$(\forall e \in E_m \cdot \neg mwr(e)) \implies (\forall e \in E_i \cdot \neg mwr(e)) . \tag{۱۱.۴}$$

از سویی دیگر با توجه به توابع انتقال دو تحلیل نتیجه می‌گیریم که هر دو تحلیل ترتیب رخدادهای مربوط به یک ریسه (ترتیب برنامه) را حفظ می‌کند، و نیز نوع رخدادهای تولید شده توسط دو تحلیل با هم برابر است. همچنین، با توجه تعریف رابطه ord در بخش ۴.۳.۴، و نیز با توجه به این نکته که مکان محاسبه شده برای رخدادها (و در نتیجه شئ‌های مربوط به رخدادهای مانیتور) با هم برابر است. بنابراین داریم:

(۱۲.۴)

$$(\forall e_۱, e_۲ \in E_m \cdot conf(e_۱, e_۲) \implies ord(e_۱, e_۲)) \implies (\forall e_۱, e_۲ \in E_i \cdot conf(e_۱, e_۲) \implies ord(e_۱, e_۲)) .$$

در نهایت با توجه به ۱۱.۴ و ۱۲.۴ نتیجه می‌گیریم: $NORACE(\eta_m) \implies NORACE(\eta_i)$. ■

فصل پنجم

پیاده‌سازی و آزمایش

در این فصل ابتدا چارچوب کامپایلر Soot، به‌صورت کلی، معرفی شده است. سپس معماری نمونه اولیه ابزار کشف رقابت داده را، که بر اساس الگوریتم‌های پیشنهاد شده در این پایان‌نامه و با استفاده از چارچوب Soot پیاده‌سازی شده است، تشریح کرده‌ایم. در پایان ابزار پیاده‌سازی شده با استفاده از چندین برنامه چندریسه‌ای جاوا محک زده شده، و نتایج آزمایش با نتایج ابزارهای مختلف مقایسه شده است.

۱.۵ مقدمه

چارچوب Soot [۴۴] یک چارچوب کامپایلر برای زبان برنامه‌نویسی جاوا است که در اصل برای پیاده‌سازی الگوریتم‌های بهینه‌سازی برنامه‌ها در نظر گرفته شده است. این چارچوب اولین بار در سال ۱۹۹۹ توسط گروهی از محققان در دانشگاه مک‌گیل در کشور کانادا توسعه داده شد. از آن پس، Soot، توسط محققان حوزه طراحی و پیاده‌سازی کامپایلرها به منظور پیاده‌سازی الگوریتم‌های بهینه‌سازی و نیز انجام تحلیل‌های جریان داده مورد استفاده قرار گرفته است. این چارچوب برای استفاده عموم آزاد است، و افراد می‌توانند از طریق تارنمای یادشده در بخش مراجع به‌صورت مجانی به فایل‌های کتابخانه‌ای و حتی متن آن فایل‌ها (که به زبان جاوا نوشته شده‌اند) دسترسی پیدا کنند. مقالات آموزشی بسیاری که توسط محققان مختلف از سراسر دنیا برای کاربردهای خاص این ابزار تهیه شده است در تارنمای یاد شده قابل یافت است. در این پایان‌نامه ما از چهاچوب Soot برای انجام تحلیل‌های جریان داده استفاده می‌کنیم.

Soot را می‌توان به عنوان یک ابزار مستقل و یا به‌صورت یک کتابخانه و چارچوبی برای پیاده‌سازی کامپایلرهای بهینه‌سازی کننده استفاده کرد. در انتخاب اول از آن برای تولید کدهای میانی به منظور بهینه‌سازی دستی استفاده می‌شود. همچنین، آن را می‌توان برای استخراج اطلاعاتی مفید از متن برنامه‌ها مانند گراف فراخوانی، اطلاعات اشاره‌گری، و نتایج تحلیل‌های پایه مانند متغیرهای مقداردهی نشده در هر نقطه از برنامه، اطلاعات وارسی `null` بودن نتیجه هر عبارت، عبارت‌های غیرقابل دسترسی در متن برنامه‌ها، و بسیاری از اطلاعات مفید دیگر استفاده کرد که برای بهینه‌سازی خودکار برنامه‌ها یا مقاصد مختلف مورد استفاده قرار می‌گیرد. در حالتی که Soot به عنوان یک چهاچوب کامپایلر استفاده می‌شود، فایل‌های کتابخانه‌ای در قالب JAR در نظر گرفته شده است که می‌توان بدون نیاز به پیاده‌سازی و یا حتی اطلاع از نحو، شیوه تحلیل

نحوی، وارسی نوع، بهینه‌سازی، و تولید کد ماشین (مجازی) برای زبان جاوا کامپایلر دلخواه خود را پیاده‌سازی کرد. برای این کار کافی است از کلاس‌های موجود در چند فایل کتابخانه‌ای برای ساخت یک یا چند بخش دلخواه از یک کامپایلر برای زبان جاوا (نسخه ۵)، بدون نیاز به پیاده‌سازی بخش‌های دیگر، استفاده کرد. برای مثال می‌توان کامپایلری را با استفاده از این چارچوب پیاده‌سازی کرد که ضمن ترجمه فایل جاوا به کد ماشین، در صورت وجود عبارت‌هایی که در آن‌ها به محتوای یک اشاره‌گر تهی صورت گرفته است را به برنامه‌نویس گزارش دهد. برای ساخت چنین کامپایلری فرد نیاز به اطلاع کامل از نحو زبان جاوای نسخه ۵، نحوه پیاده‌سازی و عملکرد تحلیل نحوی، وارسی نوع، و یا مولد کد ماشین ندارد. تنها کاری که باید انجام شود جایگزینی پیاده‌سازی یک بخش از پیش تعیین شده Soot برای این مقاصد است. در فراهم کردن اطلاعات پیاده‌سازی جدید می‌توان از اطلاعات اشاره‌گری و یا گراف فراخوانی برای ساخت یک تحلیل دقیق بین رویه‌ای استفاده کرد. به این ترتیب، Soot یک چارچوب برای ساخت کامپایلر برای زبان جاوا فراهم می‌کند، به‌طوری که افراد با صرف وقت کم می‌توانند ایده‌های خود را برای کل زبان برنامه‌سازی جاوا آزمایش کنند.

بهینه‌سازی و تحلیل در Soot بر روی نمایش‌های میانی فراهم شده توسط این چارچوب انجام می‌شود. در حالت کلی چهار نوع مختلف نمایش میانی توسط Soot ارائه شده است که پر استفاده‌ترین آن‌ها نمایش Jimple است، و ما نیز در این پایان‌نامه تحلیل‌های خود را بر حسب این شیوه از نمایش پیاده‌سازی کرده‌ایم. Jimple یک نمایش میانی نوع‌دار سه‌آدرسی برای برنامه‌های جاوا است. این نمایش به شکل دستورالعمل‌های انتساب و ۱۴ نوع دیگر دستورالعمل است که کار بر روی این نمایش به مراتب ساده‌تر از در نظر گرفتن بیش از ۲۰۰ نوع دستور در نمایش بایت‌کد جاوا و یا درگیر شدن با اصطلاحات پیچیده برنامه‌سازی و نحو گسترده زبان جاوا است. همچنین، سه‌آدرسی بودن دستورالعمل‌ها ما را قادر می‌سازد که به‌صورت یک شکل با آن‌ها رفتار کرده و بتوانیم به راحتی تحلیل‌های جریان داده خود را بیان کنیم. اطلاعات نوع متغیرها، در صورتی که از ترجمه خودکار Soot برای تبدیل برنامه‌ها جاوا به این نمایش استفاده شود، در کنار هر دستورالعمل حاشیه‌نویسی می‌شوند. این حاشیه‌نویسی متغیرها در بسیاری از کاربردها (از جمله کاربرد در این پایان‌نامه) اطلاعات مفیدی برای ساخت تحلیل‌های خبره در اختیار طراح قرار می‌دهد.

دستورالعمل‌های Jimple در Soot با استفاده از واسط `Unit` مدل می‌شوند. همان‌طور که پیش‌تر نیز اشاره کردیم، Jimple دارای ۱۵ نوع دستورالعمل است که با زیرواسط‌های `Unit` مدل شده‌اند. از مهم‌ترین این دستورالعمل‌ها می‌توان به موارد زیر اشاره کرد:

- دستورالعمل‌های بدون اثر، مقداردهی اولیه صریح، و انتساب که به ترتیب با استفاده از واسط‌های `NopStmt`، `IdentityStmt`، و `AssignStmt` در Soot مدل‌سازی می‌شود.
- دستورالعمل‌های کنترل جریان درون‌رویه‌ای مانند دستور پرش شرطی و پرش بدون شرط که در Soot به ترتیب با استفاده از واسط‌های `IfStmt` و `GotoStmt` مدل می‌شوند.
- دستورالعمل‌های کنترل جریان بین رویه‌ای مانند فراخوانی متد (مجازی و یا ایستا) و دو نوع دستورالعمل برگشت از متد (با مقدار و یا Void) که در Soot با استفاده از واسط‌های `InvokeStmt`، `ReturnStmt`، و `ReturnVoidStmt` مدل می‌شوند.
- دستورالعمل‌های همگام‌سازی با استفاده از ورود به مانیتور و خروج از آن که در Soot به ترتیب به وسیله واسط‌های `EnterMonitorStmt` و `ExitMonitorStmt` مدل می‌شوند.

- دستورالعمل‌های مدیریت استثنائات مانند دستورالعمل تولید یک استثناء که در Soot با استفاده از واسط ThrowStmt مدل می‌شوند.

این واسط‌ها و تمامی کلاس‌هایی که آن‌ها را برای نمایش Jimple پیاده‌سازی می‌کنند در پکیج soot.jimple.internal.* قابل دسترسی هستند.

با توجه به کم بودن طیف دستورات و سه‌آدرسی بودن آن‌ها، نمایش Jimple مناسب برای بسیاری از تحلیل‌ها است، همچنین، اطلاعات نوع در کنار هر دستورالعمل که به‌صورت خودکار توسط چارچوب Soot استنتاج می‌شوند، می‌توانند ساخت تحلیل‌های پیچیده را بسیار ساده‌تر کنند. همان‌طور که پیش‌تر نیز اشاره کردیم، در این پایان‌نامه، برای پیاده‌سازی تحلیل‌ها از نمایش Jimple استفاده شده است. از سایر نمایش‌های Soot می‌توان به Baf، Shimple، و Grimp اشاره کرد که هر یک مناسب برای دستیابی به یک هدف هستند. جهت آشنایی بیشتر با Soot، نمایش‌های مختلف آن، و نیز استفاده از آن به عنوان یک ابزار مستقل بهینه‌سازی و استخراج اطلاعات از متن برنامه‌ها پیشنهاد می‌کنیم که به گزارش آموزشی [۴۵] مراجعه شود.

Soot بر اساس کلاس‌های فهرست شده در زیر استوار است:

- کلاس تک عنصری Scene برای مدل‌سازی تمام محیطی که تحلیل در آن انجام می‌شود به کار می‌رود. با استفاده از آن طراح تحلیل می‌تواند کلاس‌هایی که قرار است تحلیل شود را به سیستم معرفی کند. همچنین، در طول تحلیل از طریق آن می‌توان به اطلاعات بین‌رویه‌ای مانند اطلاعات اشاره‌گری و گراف فراخوانی دست پیدا کرد.
- کلاس SootClass برای نشان دادن یک کلاس (از برنامه‌ای که در حال تحلیل آن هستیم) که به Soot معرفی شده و یا در آن ایجاد گردیده مورد استفاده قرار می‌گیرد.
- کلاس SootMethod برای نشان دادن یک متدی از یک کلاس به‌کار می‌رود.
- کلاس SootField برای نشان دادن یک فیلد از یک کلاس مورد استفاده قرار می‌گیرد.
- کلاس مجرد Body برای نشان دادن بدنه یک متد مورد استفاده قرار می‌گیرد که بر اساس شیوه نمایش بدنه متدها از زیر کلاس‌ها آن استفاده می‌شود. برای مثال، کلاس JimpleBody برای نشان دادن بدنه متدی که با استفاده از نمایش میانی Jimple فراهم شده است به‌کار می‌رود.

این کلاس‌ها و تمامی اجزای سیستم بزرگ Soot با استفاده از روش‌های طراحی شئ‌گرا در مهندسی نرم‌افزار طراحی شده‌اند، و با وجود مستندات جامعی که در تارنمای رسمی آن قابل دسترسی است به کار گیری این چارچوب ساخت کامپایلر ساده شده است.

کلاس مجرد Body واسطی برای دریافت اطلاعات سودمند در مورد بدنه متدها فراهم کرده است. برای مثال، با استفاده از متد ()getLocals می‌توان به گردایه تمام متغیرهای محلی تعریف شده در بدنه یک متد (شامل پارامترهای آن) دسترسی پیدا کرد. همچنین، به کمک متد ()getUnits می‌توان گردایه تمام دستورالعمل‌های تشکیل دهنده بدنه یک متد را دریافت کرد.

دستورالعمل‌ها در Soot، در حالت کلی، با استفاده از واسط Unit مدل شده‌اند، که بر حسب نمایش‌های مختلف دارای زیرواسط‌های اختصاصی‌تر هست. برای مثال برای نشان دادن دستوات Jimple از واسط Stmt و برای نشان دادن

دستورات Grimp از واسط `Inst` استفاده می‌شود. با استفاده از واسطی که `Unit` فراهم کرده است می‌توان از طریق متد `()getUseBoxes` به اطلاعات مختلفی مانند گردایه ساختارهایی (مقادیر ثابت، متغیرها، فیلدها، و غیره) که در یک دستورالعمل استفاده می‌شوند دست پیداد کرد. به طور مشابه با استفاده از متد `()getDefBoxes` می‌توان گردایه ساختارهایی که در یک دستورالعمل تعریف (انتساب مقدار، یا مقداردهی اولیه صریح) می‌شوند استخراج کرد. همچنین، متدهایی برای اطلاع از سایر دستورالعمل‌هایی که با دستورالعمل مورد نظر، از طریق گراف جریان کنترل، مرتبط هستند در نظر گرفته شده‌اند.

ساختارهای زبانی نمایش Jimple مانند ثوابت، متغیرهای محلی (شامل پارامترها)، و عبارات در Soot به‌ترتیب با استفاده از واسط‌های `Constants`، `Local`، و `Expr` مدل‌سازی شده‌اند. واسط `Expr` زیرواسط‌های اختصاصی‌تر برای هر دسته از عبارات دارد. تمامی این واسط‌ها و کلاس‌های پیاده‌سازی کننده آن‌ها از واسط `Value` مشتق می‌شوند.

۱.۱.۵ مراحل عملکرد Soot

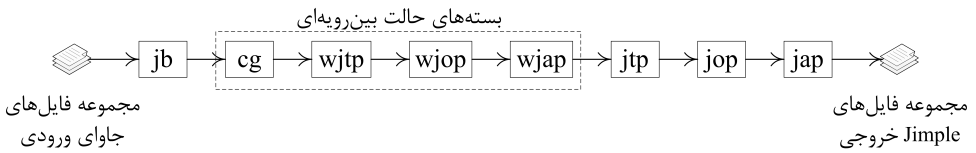
اجرای Soot شامل چندین مرحله است که هر یک از آن‌ها یک بسته یا Pack می‌گویند. اجرای Soot با تولید کد Jimple از روی کد جاوا و ارسال آن به بسته‌های دیگر شروع می‌شود. این کار را بسته‌ای به نام jb[1] با تحلیل نحوی فایل ورودی جاوا و تولید کد Jimple از روی آن انجام می‌دهد.

قاعده نام‌گذاری بسته‌ها از یک الگوی ساده پیروی می‌کند. اولین حرف نشان دهنده نمایش میانی است که بسته مورد نظر با آن سروکار دارد: حرف j برای فرم Jimple، حرف b برای فرم Baf، حرف s برای فرم Shimple، و حرف g برای فرم Grimp. حرف دوم از نام یک بسته نشان دهنده کاری است که در آن بسته انجام می‌شود: حرف b (کوتاه شده واژه Body) برای تولید بدنه متدها در یک فرم خاص، حرف t (کوتاه شده واژه Transformation) برای تحلیل‌ها و تبدیل‌های تعریف شده توسط کاربر Soot، حرف o (کوتاه شده واژه Optimization) برای بهینه‌سازی، و در نهایت حرف a (کوتاه شده واژه Attribute) برای انساب خصیصه و حاشیه‌نویسی دستورالعمل‌ها کاربرد دارد. یک حرف p در انتهای نام نیز کوتاه شده واژه Pack است، و نشان دهنده این است که شناسه مورد نظر به یک بسته تعلق دارد. برای مثال، jap (کوتاه شده Jimple Annotation Pack) حاوی تمام تحلیل‌های درون‌رویه‌ای Soot است که برای حاشیه‌نویسی کدهای تولید شده توسط سیستم بر اساس اطلاعات بدست آمده از تحلیل‌های مورد نظر به‌کار می‌رود.

در صورتی که امکان تحلیل بین‌رویه‌ای Soot فعال شود (از طریق دستور خط فرمان `w-`) گروهی دیگر از بسته‌ها فعال می‌شوند. یکی از این بسته‌ها cg (کوتاه شده Call Graph) نام دارد که وظیفه ساخت گراف فراخوانی برنامه ورودی از روی نمایش Jimple (یا هر نمایشی درونی دیگر Soot) آن را دارد. سه بسته دیگر نیز به همراه cg فعال می‌شوند، این بسته‌ها وظایفی مشابه با معادل دورن‌رویه خود دارند و نام آن‌ها با یک حرف w (کوتاه شده Whole) شروع می‌شود. تفاوت اساسی بین بسته‌های حالت بین‌رویه‌ای با معادل درون‌رویه‌ای آن‌ها در قابل دسترس بودن اطلاعات تولید شده در این بسته‌ها از تمام بسته‌های Soot از طریق `Scene` است. برای مثال نتیجه پردازش هر متد، حین پردازش متدی دیگر، قابل دسترسی است.

بسته‌هایی که در این پایان‌نامه برای ما از اهمیت ویژه‌ای برخودار هستند، بسته‌هایی است که مربوط به تبدیلات و تحلیل‌های تعریف شده توسط کاربر مربوط می‌شوند، که برای فرم Jimple این بسته jtp (کوتاه شده Jimple Transformation Pack) نام دارد. هر تبدیل یا تحلیل تعریف شده توسط کاربر با جایگزین کردن پیاده‌سازی تحلیل مورد نظر با پیاده‌سازی

[1] Body Jimple



شکل ۱.۵: نمایشی ساده شده از بسته‌های Soot و ارتباط آن‌ها با همدیگر

پیش‌فرض آن در Soot تزریق شده، و به عنوان بخشی جدید از سیستم معرفی می‌شود. شکل ۱.۵ مراحل مختلف پردازش یک برنامه جاوای ورودی در Soot و ارتباط این مراحل با همدیگر را نشان می‌دهد.

۲.۱.۵ چارچوب تحلیل جریان داده در **Soot**

طراحی تحلیل‌های جریان داده، با توجه به اینکه نمونه‌هایی از چارچوب یکنوا هستند، توسط یک رویه چهار مرحله‌ای انجام می‌شود. این مراحل به شرح زیر است:

۱- تعیین جهت تحلیل، یعنی اینکه تحلیل مورد نظر یک تحلیل رو به جلو است یا رو به عقب.

۲- تعیین قطعیت تحلیل، که در حالت کلی دو نوع قطعیت وجود دارد: قطعیت may که با انجام عمل الحاق توری برای ادغام مجموعه داده‌ی ورودی به یک گره در گراف جریان کنترل شناسایی می‌شود، و قطعیت must که با انجام عمل اشتراک توری برای ادغام مجموعه داده‌ی ورودی به یک گره در گراف جریان کنترل شناسایی می‌شود.

۳- تعیین تابع انتقال برای هر نوع بلوک. همان‌طور که در فصل ۲ مشاهده کردیم بلوک‌ها یا همان بلوک‌های ابتدایی دستورالعمل‌های برچسب‌داری هستند که گره‌های گراف جریان کنترل را تشکیل می‌دهند.

۴- تعیین مقدار اولیه برای گره ورودی گراف جریان کنترل. لازم به یادآوری است که در Soot علاوه بر این کاربر نیاز دارد که مقدار اولیه‌ای برای تک تک گره‌های گراف اتخاذ کند.

همان‌طور که پیش‌تر مشاهده کردیم، در انجام تحلیل جریان داده نیاز داریم که برنامه ورودی را به نحوی نمایش دهیم که روش پیشنهاد شده در بسیاری از مراجع استفاده از گراف جریان کنترل است. گراف جریان کنترل یک گراف جهت‌دار بوده و در Soot با استفاده از واسط `DirectedGraph` نمایش داده می‌شود. این واسط از طریق بسته `soot.toolkits.graph.*` قابل دسترسی است.

گام اول: تعیین جهت تحلیل. Soot امکان پیاده‌سازی تحلیل جریان داده در هر دو جهت رو به جلو و رو به عقب را فراهم کرده است. کلاس‌های مجرد `ForwardFlowAnalysis` و `BackwardFlowAnalysis` به ترتیب برای پیاده‌سازی تحلیل‌های رو به جلو و رو به عقب به کار می‌رود. برای این کار طراح تحلیل کلاسی از این کلاس‌ها مشتق کرده و متدهای مجرد آن را پیاده‌سازی می‌کند. سازنده چنین کلاسی باید یک گراف جهت دار را از ورودی دریافت کند و قبل از هر کاری آن را به سازنده کلاس والد خود پاس کند، در انتها نیز متد (به ارث برده شده) `()doAnalysis` فراخوانی نماید. هر دوی این کلاس‌ها زیرکلاس‌های کلاس `FlowAnalysis` هستند. این کلاس‌ها را می‌توان از بسته `soot.toolkits.scalar.*` دریافت کرد.

گام دوم: تعیین قطعیت تحلیل. در این مرحله طراح تحلیل تصمیم می‌گیرد که ادغام دو عنصر از توری تشکیل دهنده فضای اطلاعات تحلیل را چگونه انجام دهد. در صورتی که این کار با عملگر الحاق توری انجام شود گوییم یک تحلیل با قطعیت may داریم، و اگر عمل ادغام عناصر با عملگر اشتراک انجام شود گوییم یک تحلیل با قطعیت must داریم. در تحلیل‌های جریان داده، در حالت کلی، عمل ادغام زمانی انجام می‌شود که بیش از یک کمان جریان کنترل وارد یک گره می‌شود. در این حالت مجموعه داده‌ای که از هر یک از این کمان‌ها به‌دست می‌آید، برای محاسبه برآیند داده، با هم ادغام می‌شوند. Soot فرض می‌کند که عملگر الحاق توری مورد نظر طراح تحلیل دارای خواص شرکت‌پذیری و جابجایی‌پذیری باشد. بنابراین، فرض بر این است که می‌توان کوچکترین کران بالا یا بزرگترین کران پایین یک مجموعه از داده را می‌توان با یک عملگر ادغام دوتایی ادغام کرد. طراح تحلیل برای پیاده‌سازی عملگر ادغام مورد نظر خود، باید متد مجرد `merge` از کلاس `FlowAnalysis` را پیاده‌سازی کند. این متد سه پارامتر دارد، که باید دو پارامتر اول را با هم ادغام و در پارامتر سوم قرار دهیم.

از آنجایی که ممکن است برای برخی از کاربردها حتی کپی گرفتن از روی یک عنصر از توری کار پیچیده‌ای باشد، Soot متد مجرد `copy` از کلاس `FlowAnalysis` را در اختیار طراح قرار می‌دهد که با پیاده‌سازی آن کار کپی کردن عناصر توری را بر عهده بگیرد. این متد دارای دو پارامتر است، و چارچوب انتظار دارد که با فراخوانی آن متد یک ارجاع به یک کپی از آرگومان اول را در آرگومان دوم مشاهده کند.

گام سوم: تعیین تابع انتقال. در این گام اصلی‌ترین کار تحلیل، یعنی تابع انتقال برای بلوک‌های ابتدایی، تعریف می‌شود. چارچوب Soot در هر گره از گراف فراخوانی تابع انتقال را برای گره مورد نظر فراخوانی می‌کند تا با ارائه داده ورودی گره، داده خروجی آن را محاسبه کند. متد مجرد `flowThrough` از کلاس `FlowAnalysis` برای این کار در نظر گرفته شده است. این متد دارای سه پارامتر است: پارامتر اول داده ورودی بلوک ابتدایی را مشخص می‌کند، پارامتر دوم ارجاعی است به بلوک ابتدایی مورد نظر، و در نهایت پارامتر سوم برای قرار دادن نتیجه محاسبات در نظر گرفته شده است.

گام چهارم: تعیین مقادیر اولیه تحلیل. در این گام مقدار اولیه برای ورودی مدخل گراف جریان کنترل تعیین می‌شود. همچنین، در چارچوب Soot برای سادگی در پیاده‌سازی الگوریتم محاسبه پاسخ تحلیل، نیاز داریم که یک مقدار اولیه برای ورودی هر یک از گره‌های گراف جریان کنترل (به غیر از گره مدخل) تعیین کنیم. چارچوب Soot انتظار دارد که مقدار اولیه برای ورودی گره مدخل گراف جریان کنترل توسط متد مجرد `entryIntialFlow` از کلاس `FlowAnalysis` برگشت داده شود. مقدار اولیه برای سایر گره‌ها نیز توسط متد مجرد `newInitialFlow` از همان کلاس تأمین می‌شود. طراح تحلیل باید این دو متد مجرد را به دلخواه خود پیاده‌سازی کند.

Soot چندین نوع گراف جهت‌دار برای فراهم کردن گراف جریان کنترل پیاده‌سازی کرده است. مناسب‌ترین پیاده‌سازی برای کاربرد ما در این پایان‌نامه کلاس `ExceptionalUnitGraph` است، که گره‌های آن اشیائی از زیرکلاس‌های `Unit` است و کمان‌های آن با استفاده از مجموعه‌ای از دوتایی‌ها از این گره‌ها پیاده‌سازی شده است. تفاوت این نوع گراف‌ها با سایر گراف‌های جریان کنترل در این است که در اینجا جریان‌های کنترلی حاصل از استثنائات نیز در نظر گرفته می‌شود.

۳.۱.۵ جمع‌بندی

Soot یک چارچوب پیاده‌سازی کامپایلر برای زبان برنامه‌سازی جاوا است که با فراهم کردن مجموعه‌ای از کلاس‌های کتابخانه‌ای، به زبان جاوا، امکاناتی برای سفارشی سازی مراحل کامپایل ایجاد کرده است. کاربر Soot می‌تواند کارهای مختلفی از جمله انجام تحلیل‌های جریان داده و نیز بهینه‌سازی را با سفارشی سازی بخش‌های مختلف چارچوب انجام دهد. این ابزار هزینه پیاده‌سازی کامپایلرها را بسیار پایین می‌آورد، به‌گونه‌ای که تنها با چند صد خط کد جاوا می‌توان آن را برای ساخت یک کامپایلر جاوای دلخواه تنظیم کرد. مطالب ارائه شده در این فصل با هدف ارائه یک حس از امکانات Soot که در این پایان‌نامه استفاده کرده‌ایم تدوین شده است، حال آنکه امکانات Soot فراتر از نیازهای پیاده‌سازی موجود این پایان‌نامه است. گزارش آموزشی [۴۵] نقطه شروع مناسبی برای یادگیری عمیق‌تر این ابزار است.

۲.۵ پیاده‌سازی

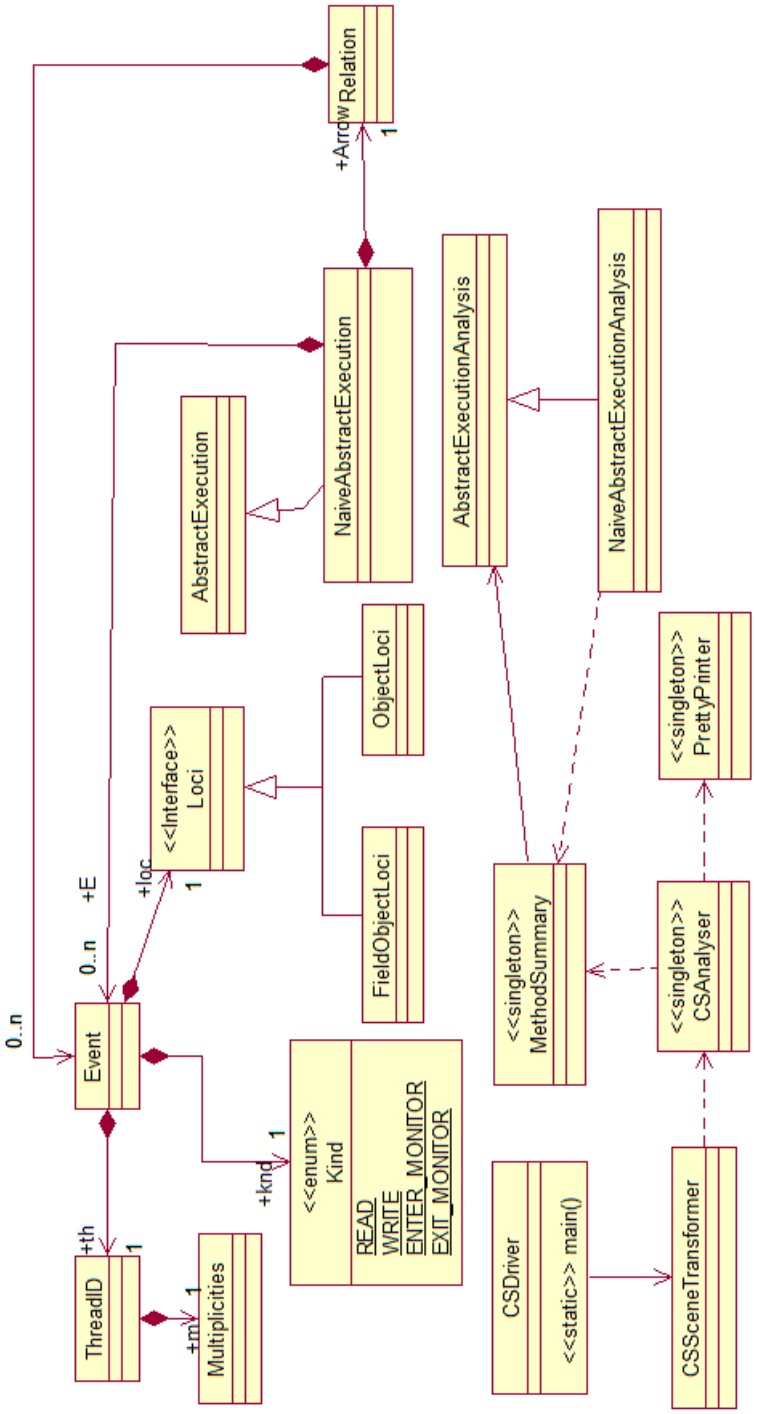
ایده‌های مربوط به کشف رقابت داده که در این پایان‌نامه ارائه داده‌ایم با استفاده از چارچوب Soot پیاده‌سازی شده است. خوشبختانه از آنجایی چارچوب Soot خود تحلیل‌های گراف فراخوانی و نیز اشاره‌گر دقیقی را فراهم می‌کند، نیازی به پیاده‌سازی مجدد این تحلیل‌ها نداریم. می‌توانیم با فعال سازی بسته‌های مربوط به تحلیل‌های بین‌رویه‌ای به‌صورت خودکار تحلیل گراف فراخوانی درونی Soot را فعال و از طریق `Scene` به اطلاعات محاسبه شده توسط آن دست پیدا کرد. همچنین، تحلیل اشاره‌گر درونی Soot که Spark نام دارد را می‌توان از طریق آرگومان خط فرمان `-p cg.spark enabled : true` فعال کرد. اطلاعات محاسبه شده توسط این تحلیل نیز با استفاده از `Scene` قابل دسترسی است.

بدنه اصلی پیاده‌سازی انجام شده را در بسته wjtp نصب کرده‌ایم، که وظیفه انجام تحلیل اجرای مجرد را بر عهده دارد. یک نمودار کلاس به راحتی می‌تواند ساختار کلی سیستم پیاده‌سازی شده را نمایش دهد. نمودار کلاس برای این سیستم ارائه در شکل ۲.۵ قابل مشاهده است، این نمودار می‌تواند راهنمای بررسی متن ابزار پیاده‌سازی قرار گیرد.

۱.۲.۵ آزمایش

در این بخش نتایج ارزیابی ابزار کشف رقابت داده پیاده‌سازی شده بر روی چندین برنامه محک جاوا ارائه می‌گردد، در ادامه با مشاهده نتایج حاصل از آزمایش الگوریتم‌های پیاده‌سازی شده متوجه می‌شویم که دقت این ابزار کشف رقابت داده درست و پیمانه‌ای قابل مقایسه با دقیق‌ترین ابزارهای کشف رقابت داده است.

در حالت کلی موارد رقابت داده که توسط این ابزار گزارش می‌شود به دو دسته تقسیم می‌شود: ۱- رقابت‌هایی که از ریسه‌های چندگانه گزارش می‌شود، ۲- رقابت‌هایی که از دو ریسه همروند گزارش می‌گردند. برای مقایسه ابزار پیاده‌سازی شده با سایر ابزارهای کشف رقابت داده، نتیجه تولید شده توسط آن نیاز به یک پس‌پردازش برای حذف پیام‌های خطای بی‌مورد و تکراری است. ما این کار را تا حدی در پیاده‌سازی و قبل از نمایش نتیجه نهایی تحلیل انجام داده‌ایم، اما این صافی خودکار پیام‌های خطا به اندازه کافی هوشمند نیست که بتواند پیام‌های خطای متعدد را به یک شئ (که رقابت داده گزارش شده بر روی آن اتفاق می‌افتد) ربط دهد. بنابراین، در شمارش تعداد خطاهای رقابت داده به ازای تمام موارد رقابت داده گزارش شده مربوط به یک شئ که توسط یک جفت ریسه به وجود می‌آید یک رقابت داده در نظر گرفته‌ایم. همچنین، به ازای هر مورد رقابت داده گزارش شده مربوط به یک شئ که توسط یک ریسه چندگانه به وجود می‌آید یک رقابت داده در



شکل ۲.۵: نمودار کلاس ابزار پیاده‌سازی شده برای کشف رقابت داده

نظر گرفته‌ایم. برای مقایسه با سایر ابزارها که شیوه ارائه نتیجه خطاها به سیستم ما شباهت دارد چنین پس پردازش دستی انجام نشده است.

در جدول ۱.۵ مشخصات برنامه‌هایی را فهرست کرده‌ایم که برای محک سیستم پیاده‌سازی شده مورد استفاده قرار گرفته است. این برنامه‌ها به‌گونه‌ای انتخاب شده‌اند که اولاً طیف وسیعی از اصطلاحات برنامه‌سازی همروند در آن‌ها استفاده شده

باشد، و ثانیاً پیش‌تر توسط تعداد زیادی از محققان به عنوان مجموعه برنامه‌های محک مورد استفاده قرار گرفته باشند. بنابراین، با استفاده از آنها قادر به انجام یک مقایسه مؤثر می‌شویم. آزمایش‌ها را در یک کامپیوتر شخصی دارای سیستم عامل لینوکس نسخه ۳/۱۳، با ۴ گیگابایت حافظه اصلی، و ۲/۵ گیگاهرتز پردازنده انجام داده‌ایم. در نهایت، اندازه فضای هیپ اخذ شده در ماشین مجازی برای اجرای ابزار ۲/۵ گیگابایت است. جزئیات آزمایش مربوط به زمان اجرا و نیز تعداد گزارشات خطا در جدول ۲.۵ فهرست شده است.

جدول ۱.۵: مشخصات مجموعه برنامه‌های چندریسه‌ای جاوا که برای محک ابزار کشف رقابت داده استفاده شده است (چهار برنامه اول از طریق رایانامه از آقای Christoph von Praun دریافت شده است.)

برنامه محک	تعداد خطوط کد برای کل برنامه	توضیحات
Elevator	۵۳۱	یک شبیه‌ساز بلادرنگ رخدادهای گسسته (پروژه درسی در دانشگاه ETH)
Philo	۸۴	یک شبیه‌سازی از مسأله شام فیلسوفان،
Sor	۱۷۷۰۴	یک پیاده‌سازی برای حل مسأله Successive Over-Relaxation (پروژه درسی در دانشگاه ETH)،
TSP	۷۰۶	برنامه حل مسأله فروشنده دوره‌گرد (پروژه درسی در دانشگاه ETH)،
Example	۳۰	یک برنامه ساده برای نمایش تعامل دو ریسه از طریق یک شئ مشترک، که در مقاله [۲۳] به عنوان یک مثال برای توضیح بخش‌های مختلف الگوریتم کشف رقابت داده ارائه شده است،
Battleship	۲۲۹۷	یک بازی چند ریسه‌ای (پروژه درسی در دانشگاه امیرکبیر)،
Clanbomber	۲۶۳۰	یک بازی چند ریسه‌ای (پروژه درسی در دانشگاه امیرکبیر).

جدول ۲.۵: تعداد خطاهای رقابت داده که، توسط ابزار پیاده‌سازی شده در این پایان‌نامه، برای هر یک از برنامه‌های محک گزارش شده است. ستون مربوط به تعداد گزارشات خطا با پس‌پردازش، برای مقایسه ابزار ما با ابزار معرفی شده در [۲۳] اضافه شده، و تعداد موارد رقابت شئ‌ها را نشان می‌دهد. ستون دوم مدت زمانی را نشان می‌دهد که طول کشید تا ابزار جرای خود را به پایان برساند. توجه کنید که نزدیک به نیمی از این مقادیر صرف انجام تحلیل‌های درونی Soot می‌شود.

برنامه محک	مدت زمان تحلیل	تعداد گزارشات خطا بدون پس‌پردازش		تعداد گزارشات خطا با پس‌پردازش	
		جفت متضاد	نوشتن چندگانه	جفت متضاد	نوشتن چندگانه
Elevator	۱ دقیقه و ۲۰ ثانیه	۳۰	۳۴	۱	۴
Philo	۱ دقیقه	۰	۵	۰	۲
Sor	۲ دقیقه	۴	۲۰	۲	۳
TSP	۸ دقیقه و ۴۴ ثانیه	۸	۵۸	۳	۱۲
Example	۱ دقیقه	۰	۰	۰	۰
Battleship	۳ دقیقه	۰	۰	۰	۰
Clanbomber	۳۴ دقیقه و ۴۰ ثانیه	۱۰۷۲	۰	۱۴	۰

زمان لازم برای تحلیل توسط یک ابزار نقشی اساسی در مؤثر بودن آن در استفاده در صنعت دارد. به‌طور معمول این زمان رابطه مستقیمی با اندازه برنامه برحسب تعداد خطوط کد دارد. همان‌طور که در جدول ۲.۵ مشاهده می‌شود، مدت

زمان تحلیل برای بعضی از برنامه‌ها متناسب با اندازه کل برنامه نیست. برای مثال برنامه Sor با ۱۷۷۰۴ خط کد ۲ دقیقه زمان برای تحلیل نیاز دارد، حال آن‌که برنامه Clanbomber با ۲۶۳۰ خط کد به زمانی بیش از ۳۰ دقیقه برای تحلیل نیاز دارد. این پدیده را چنین توجیه می‌کنیم که چون ابزار توسعه داده شده در این پایان‌نامه هر متد از برنامه را در انزوا تحلیل می‌کند، و اصلی‌ترین پیچیدگی محاسبات مربوط به تحلیل یک متد و محاسبه جواب برای دستگاه معادلات جریان داده برای آن متد است، اندازه متدها در برنامه (برحسب تعداد خطوط کد بدنه آن‌ها) اصلی‌ترین تأثیر را در تعیین زمان تحلیل دارند. برنامه Clanbomber دارای چند متد بزرگ (بیش از ۲۰۰ خط کد) است، حال آنکه اندازه تمام متدهای برنامه Sor از ۷۰ خط تجاوز نمی‌کند. این توجیه منطقی است، چرا که با توجه به پیچیدگی محاسباتی برای بدست آوردن جواب برای یک دستگاه معادلات که برای بدنه متدی با n خط کد تشکیل شده، از مرتبه $n^۴$ است. بنابراین، عامل اصلی در تعیین زمان لازم برای تحلیل یک برنامه توسط ابزار ما اندازه بدنه متدها است، نه اندازه کل برنامه.

ما با بررسی نتایج تحلیل متوجه شدیم که در تمامی موارد آزمایش وجود یک تحلیل فرار (برای تعیین حوزه قابلیت مشاهده شدن اشیائی که توسط هر ریسه ایجاد می‌شوند) تعداد خطاهای گزارش شده را به مقدار قابل توجهی کاهش خواهد داد. در مورد برنامه Elevator خطاهای جفت متضاد مربوط به مقداردهی اولیه در سازنده کلاس `Lift` است. در اینجا عملیات نوشتن مورد نظر قبل از اینکه تنها ریسه برنامه فعال شود، انجام می‌گردد. بنابراین، از آنجایی شئ `Lift` مورد نظر هنوز در حوزه دید ریسه‌های موازی با ریسه اصلی قرار نگرفته است، خطای گزارش شده بی‌مورد بوده و یک تحلیل فرار ساده به راحتی می‌تواند این موضوع را تشخیص دهد. خطاهای نوشتن چندگانه نیز به تنها ریسه مجرد برنامه، که درون یک حلقه ایجا می‌شود، تعلق دارد. در این ریسه علاوه بر اینکه بر روی تعدادی از فیلدهای مربوط به خود شئ نوشته می‌شود (یک مورد)، بر روی فیلدهای دو شئ تازه ایجاد شده نیز نوشته می‌شود که هر دو شئ به‌صورت محلی توسط ریسه مورد نظر مورد دسترسی قرار گرفته، و سایر نمونه‌های (در زمان اجرای) ریسه مجرد یاد شده عملاً توانایی دسترسی به آن‌ها را ندارند. همان‌طور که مشاهده می‌شود، هر سه مورد خطاها را می‌توان به کمک یک تحلیل فرار ساده برطرف کرد. آخرین مورد خطای چندگانه مربوط به نوشتن بر روی فیلد `floors` از شئ‌ای از کلاس `Controls` در این برنامه است. از آنجایی که در کل برنامه تنها یک نمونه از این کلاس ایجاد می‌شود، و چون تمامی دسترسی‌ها به فیدهای این شئ از طریق متدهایی از آن کلاس صورت می‌گیرد که عمل همگام‌سازی را با استفاده از خود شئ انجام می‌دهند، گزارش چنین خطایی نیز بی‌مورد است.

در بررسی برنامه Philo دو مورد خطای نوشتن چندگانه مربوط به عملیات نوشتن تنها ریسه مجرد برنامه بر روی فیلدهای خود شئ و نیز فیلدهای شئ `Table` پاس شده به هر نمونه از ریسه مجرد مورد نظر است. خطاهای مربوط به دسترسی‌ها به فیلدهای خود بی‌مورد بوده و با انجام یک تحلیل فرار ساده بر طرف می‌شوند. مورد دیگر خطا مربوط به بروز رسانی فیلدهای تنها شئ ایجاد شده در برنامه از کلاس `Table` است. از آنجایی که در کل برنامه تنها یک نمونه از این کلاس موجود است و تمامی دسترسی‌ها توسط متدهای همگام‌سازی شده بر روی خود شئ انجام می‌شود، گزارش چنین خطایی نیز بی‌مورد است.

در بررسی برنامه Sor دو مورد خطای جفت متضاد مربوط به جفت رخدادهای ایجاد شده در متد `main` و نیز دو ریسه مجرد برنامه، بعد از فراخوانی عملگر `join` بر روی هر دوی آن‌ها است. در پیاده‌سازی یک ابزار درست، به منظور تشخیص خطاهای رقابت داده برای این برنامه، این دو مورد خطا بی‌مورد نیستند چرا که از روی متن برنامه، و حلقه تکراری که عملگر `join` را بر روی تعدادی از عناصر یک آرایه از ریسه‌ها اعمال می‌کند، قادر به تشخیص این موضوع نخواهیم بود که چه تعداد از نمونه‌های ریسه‌های مجرد در آن نقطه متوقف شده‌اند. یکی از موارد خطای نوشتن چندگانه مربوط به عملیات نوشتن بر روی فیلدهای خود شئ است، که با توجه به اینکه تمامی فیلدهای اشیاء مورد نظر از نوع پایه‌ای زبان جاوا بوده، یک تحلیل

فرار ساده ما را قادر می‌سازد که این خطا را برطرف کنیم. دو خطای دیگر مربوط به نوشتن ریسه مجرد چندگانه بر روی فیلدهای ایستا کلاس Sor است. این دو مورد خطا نیز در پیاده‌سازی یک ابزار کشف رقابت داده درست بی‌مورد نیست چرا که دسترسی‌ها توسط هیچ قفلی محافظت نشده‌اند، هرچند که با وارسی دستی برنامه متوجه می‌شویم که برنامه در حقیقت خالی از رقابت داده است.

برنامه Battleship بر خلاف بزرگی، منطق ساده‌ای دارد. ابتدا تمام عملیات برنامه توسط متدهای فراخوانی شده در متد main کلاس ServerStart انجام شده، و در نهایت ریسه‌ای ایجاد می‌گردد. خطاهای گزارش شده برای برنامه Clanbomber، بر خلاف موارد قبل، کمترین تعداد گزارشات نادرست را در خود دارد. تا آنجا که توان بررسی گزارشات خطا را داشتیم، مشاهده کردیم که گزارشات مربوط به دسترسی‌هایی می‌شود که در آن‌ها یک مکان مشترک حافظه بدون همگام‌سازی توسط دو ریسه مورد دسترسی قرار می‌گیرد. با بررسی متن برنامه متوجه می‌شویم که برنامه‌نویس از هیچ گونه مکانیزم همگام‌سازی در برنامه خود استفاده نکرده است. برای مثال، سه ریسه در سازنده کلاس Game ایجاد می‌شود، بعد از ایجاد این سه ریسه که به فیلدهایی از کلاس Main (بدون همگام‌سازی) دسترسی دارند، ریسه اصلی برنامه با ایجاد شئ‌هایی از کلاس‌های Board، Options، و MainFrame توسط تعداد زیادی از دستورالعمل‌ها در بدنه متدهای این کلاس‌ها به فیلدهای خود ایستا کلاس Main دسترسی پیدا می‌کند.

در نهایت Example، برنامه ساده‌ای است و با بررسی آن متوجه می‌شویم که خالی از رقابت داده است. این برنامه به راحتی و با یک بررسی چشمی قابل وارسی است و نیازی به توضیح احساس نمی‌شود.

برای مقایسه ابزار پیاده‌سازی شده با ابزار معرفی شده در [۲۳] لازم است به سه نکته توجه کنیم: ۱- ابزار معرفی شده در [۲۳] درست نیست، ۲- ابزار معرفی شده در [۲۳] موارد رقابت داده بر روی شئ‌ها را شمارش می‌کند نه دستورالعمل‌هایی که موجب به وجود آمدن رقابت داده می‌شوند، ۳- با توجه به اینکه ابزار معرفی شده در [۲۳] مجهز به یک تحلیل فرار ساده است، شیوه برخورد آن با ریسه‌های چندگانه متفاوت از شیوه برخورد ابزار ما با این دسته از ریسه‌ها است.

نادرست بودن و نیز شیوه شمارش خطاهای گزارش شده در ابزار معرفی شده در [۲۳] با توجه به هدف محقق در جهت ساخت یک ابزار صنعتی و مؤثر برای کاهش تعداد خطاهای ارائه شده در یک ابزار تست پویا به منظور کشف رقابت شئ [۹] قابل توجیه است. در مورد برنامه‌های آزمایش شده نادرست بودن ابزار، برای مثال در برنامه Sor ناشی از در نظر گرفتن اثر اعمال عملگر join برای تمامی عناصر آرایه است، که این باعث می‌شود خطاهای قابل گزارش در اثر دسترسی‌های بدون همگام‌سازی ریسه اصلی برنامه و دو ریسه مجرد ایجاد شده در نظر گرفته نشود. از آنجایی که ابزار پیاده‌سازی شده در این پایان‌نامه قرار است که درست باشد، قادر نیستم خطاهایی که تنها احتمال بی‌مورد بودن آن‌ها وجود دارد را در نظر نگیریم.

با انجام یک پس‌پردازش برای تبدیل تعداد خطاهای گزارش شده، که بر حسب دستورالعمل‌های برنامه هستند، به تعداد شئ‌های تحت تأثیر این دستورالعمل‌ها، خروجی ابزار خود را شبیه به خروجی ابزار معرفی شده در [۲۳] کرده‌ایم. در نهایت، لازم به یادآوری است که هدف ما از پیاده‌سازی و انجام آزمایش در این پایان‌نامه نشان دادن این موضوع است که ساخت یک ابزار ایستا پیمانه‌ای برای کشف رقابت داده به‌طوری که نتایج حاصل شده از آن قابل مقایسه با دقیق‌ترین ابزارهای کشف رقابت داده باشد دور از انتظار نیست. بنابراین، از آنجایی که تعبیه یک تحلیل فرار در ابزار پیاده‌سازی شده و نیز بهبود الگوریتم‌های استخراج اجراهای مجرد و فیلترینگ خروجی ابزار کار دشواری نیست، انتظار می‌رود که بتوان دقت این ابزار را بسیار بیشتر افزایش داد.

جدول ۳.۵ نتیجه آزمایش دو ابزار مشابه معرفی شده در مقالات [۲۰] و [۲۳] است. توجه کنید که با توجه به در دسترس نبودن ابزارهای یاد شده، قادر نبودیم برنامه‌های بیشتری را آزمایش کنیم، و یا برنامه‌هایی مانند Example را با استفاده از

ابزار Chord وارسی کنیم. سطر اول این جدول برنامه‌های مورد آزمایش را فهرست می‌کند. در هر خانه جدول تعداد کل خطاهای گزارش شده توسط هر ابزار را نوشته‌ایم. توجه کنید که علامت «؟» به معنای در دسترس نبودن نتیجه آزمایش مورد نظر است، اما با توجه به اینکه برنامه Example برنامه ساده‌ای است، پیش‌بینی می‌کنیم که این مقدار برای آن برابر با صفر باشد. همان‌طور که مشاهده می‌شود، تنها مورد اختلاف زیاد ابزار ما با ابزار معرفی شده در [۲۳] در مورد برنامه Sor

جدول ۳.۵: تعداد خطاهای رقابت داده که توسط دو ابزار معرفی شده در [۲۰، ۲۳] برای هر یک از برنامه‌های محک گزارش شده است

	Elevator	Philo	Sor	TSP	Example	Battleship	Clanbomber
Chord[۲۰]	۰	۰	؟	۴	؟	؟	؟
ابزار معرفی شده در [۲۳]	۴	۱	۱	۳	۰	؟	؟

و TSP است که با توجه به توضیحاتی که پیش‌تر ارائه شد، برای یک ابزار درست امکان صرف نظر کردن از این گزارشات وجود ندارد. ابزار Chord نیز دقیق‌ترین ابزار ارائه شده تا کنون برای کشف ایستا رقابت داده در برنامه‌های زبان جاوا است. این ابزار درست بوده، و بنابراین می‌توان از آن برای اعتبارسنجی نتایج استفاده کرد. خروجی ابزار Chord همانند ابزار ما بر حسب جفت دستورالعمل‌ها است، در نتیجه می‌توان بدون هیچ پس‌پردازشی میزان مؤثر بودن دو ابزار را با هم مقایسه کرد. اما همان‌طور که مشاهده می‌شود اختلاف فاحشی بین نتایج دو ابزار وجود دارد (در مورد برنامه Elevator، ۶۴ خطا در مقابل صفر خطا). ما علت این اختلاف را وجود تحلیل فرار و نیز حساس به زمینه بودن تحلیل دگرنامی در Chord می‌دانیم. تحلیل فرار به‌کار گرفته شده در Chord، با توجه به آزمایشات صورت گرفته بر روی ۱۲ برنامه بزرگ، به‌طور متوسط بیش از ۳۰۰۰ جفت دستورالعمل محتمل در شرکت در رقابت داده را از دور بررسی بیشتر خارج می‌کند، همچنین، وجود تحلیل دگرنامی حساس به زمینه در Chord به عنوان یک عامل مهم در بهبود دقت ابزار معرفی شده است [۱۹].

نقطه قوت تحلیل ارائه شده در این پایان‌نامه پیمانه‌ای بودن آن است. پیمانه‌ای بودن به این معنا است که هر متد برنامه در انزوا تحلیل می‌شود، و نیازی به کل برنامه نیست. این یعنی بر خلاف ابزاری مانند Chord پیچیده‌ترین مرحله محاسبات، که محاسبه جواب برای دستگاه معادلات جریان داده است، برای کل برنامه انجام نمی‌شود بلکه برای هر متد از برنامه به‌صورت جداگانه جوابی پارامتری محاسبه شده و در محل‌های فراخوانی، متناسب با زمینه فراخوانی، نمونه‌سازی می‌شود. تنها مشکل نمونه اولیه ابزار پیاده‌سازی شده در این پایان‌نامه (به غیر از نبود دقت کافی به دلیل تعبیه نکردن تحلیل فرار و تحلیل اشاره‌گر دقیق‌تر) نبود روشی مناسب برای ذخیره و بازیابی خلاصه متدها است. این ابزار خلاصه تمام متدهای برنامه را در حافظه اصلی (در هیپ برنامه) نگه می‌دارد، که این باعث می‌شود این حافظه برای برنامه‌های بزرگ کافی نباشد. این مشکل را به راحتی می‌توان با ارائه یک روش مناسب ذخیره و بازیابی اطلاعات که در هر لحظه حداکثر به اندازه یک مقدار ثابت از خلاصه متدها را در حافظه هیپ نگه‌داری و بقیه اطلاعات را در حافظه‌های بزرگتر جانبی ذخیره می‌کند، حل کرد. با این حال ابزار پیاده‌سازی شده برای برنامه‌های بزرگ مقیاس‌پذیرتر از ابزارهایی مانند Chord است، چرا که با بررسی نتایج آزمایش این ابزار بر روی برنامه‌های بزرگتر از ۴ هزار خط کد مشاهده می‌کنیم که ابزار یاد شده قادر به تحلیل برنامه‌هایی به این بزرگی را ندارد [۲۴]. مشکل سرعت تحلیل را نیز با به‌کارگیری ساختمان‌داده‌های بهتر به جای استفاده از مجموعه‌ها (که حتی عملیات ساده بر روی آن‌ها زمان‌بر است) می‌توان برطرف نمود. از آنجایی که در این پایان‌نامه یک تحلیل اشاره‌گر حساس به زمینه پیمانه‌ای ارائه شده است، تعبیه چنین تحلیلی در ابزار ارائه شده دور از انتظار نیست. همچنین، تحلیل اشاره‌گر ارائه شده در [۳۶] نشان می‌دهد که می‌توان اطلاعات فرار در مورد شئ‌ها را در گراف شکل هیپ گنجاند، به‌طوری

که یک تحلیل اشاره‌گر پیمانه‌ای به موازات یک تحلیل فرار پیمانه‌ای ساخت.

مقاله [۳۲] نیز ابزاری دیگر (به نام Houdini/rcc) برای کشف ایستا رقابت داده ارائه می‌دهد. هسته این ابزار یک زبان جاوای گسترش یافته (به نام rccjava) است، که در آن از یک سیستم نوع برای پیش‌گیری از رقابت داده استفاده می‌شود. یک سیستم جانبی (به نام Houdini) به‌منظور حدس حاشیه‌نوشته‌های لازم برای سیستم نوع به‌کار گرفته می‌شود. متأسفانه ابزار مورد نظر به‌صورت یک فایل اجرایی موجود نبود و کامپایل و راه اندازی آن نیز میسر نشد. همچنین، برنامه‌هایی که برای محک ابزار مورد نظر استفاده شده بود در دسترس نیست. بنابراین، به یک مقایسه سطحی بسنده می‌کنیم. ابزار Houdini/Rcc به‌طور متوسط حدود ۱۷٫۹ خطا برای هر ۱۰۰۰ خط کد جاوا تولید می‌کند، این در حالی است که ابزار یاد شده پیمانه‌ای نیست و به‌طور متوسط حدود ۶۷٫۹ دقیقه زمان برای تحلیل یک برنامه لازم دارد. ابزار ابتدایی پیاده‌سازی شده در این پایان‌نامه حدود ۵۱٫۳ خطا در هر ۱۰۰۰ خط کد گزارش داده و زمانی در حدود ۷٫۳ دقیقه برای انجام تحلیل لازم دارد.

فصل ششم

تحلیل جریان کنترل و اشاره‌گر

در این فصل ابتدا یک تحلیل جریان کنترل، برای پیش‌بینی جریان کنترل در محل‌های فراخوانی متدها معرفی می‌شود. سپس یکی از تحلیل‌های اساسی به‌کار رفته در پایان‌نامه، یعنی تحلیل اشاره‌گر ارائه می‌گردد.

در تحلیل جریان کنترل ارائه شده در این پایان‌نامه قصد داریم از روش‌های معمول و مرسوم برای محاسبه گراف فراخوانی استفاده کنیم. این تحلیل بر اساس تحلیل سلسله مراتب کلاس (CHA)، و نیز تحلیل سریع نوع (RTA) است. برای آشنایی بیشتر با این تحلیل‌ها خواننده می‌تواند به مقاله [۴۶] مراجعه کند. همان‌طور که پیش‌تر نیز اشاره کرده‌ایم، نتیجه تحلیل جریان کنترل ارائه شده در این فصل، در نهایت، گراف فراخوانی متدها است که می‌توان اطلاعاتی مربوط به روابط فراخوانی‌کننده-فراخوانی‌شونده بین متدها، و نیز اطلاعات جریان کنترل در محل‌های فراخوانی متدها را از آن استخراج کرد.

ایده‌های اولیه تحلیل اشاره‌گر طراحی شده در این پایان‌نامه نیز بر اساس تحلیل شکل هیپ و اشاره‌گر ارائه شده در دو مقاله [۴۷، ۳۶] است. تحلیل ارائه شده پیمانه‌ای است، و قابلیت تحلیل مجزای بخش‌های مختلف برنامه‌ها را دارد. این تحلیل همچنین قابلیت استخراج اطلاعات چندگانگی اشیاء را دارد. در نهایت با توجه به حساس به متن بودن تحلیل انتظار داریم کیفیت اطلاعات بدست آمده بالا باشد. برای بالا بردن هر چه بیش‌تر دقت اطلاعات بدست آمده از تحلیل متدهایی که هنوز متدهای فراخوانی کننده آن‌ها در دسترس نیست، حاشیه‌نوشته‌هایی را به نحو زبان مورد بررسی افزوده‌ایم. این حاشیه‌نوشته‌ها تنها در اعلان متدها نوشته می‌شوند. بنابراین، همانند حاشیه‌نوشته‌های ارائه شده در فصل چهارم سربار قابل توجهی را به برنامه‌نویس تحمیل نمی‌کنند. در انتها سیستم نوع استاندارد ارائه شده در پیوست اول را، به منظور صحت‌سنجی برنامه‌ها برای تبعیت از حاشیه‌نوشته‌های دگرنامی، توسعه داده‌ایم.

۱.۶ تحلیل جریان کنترل

در تحلیل جریان کنترل ارائه شده در این پایان‌نامه هر متد را تک تک و به‌صورت مجزا مورد بررسی قرار می‌دهیم. در تحلیل هر متد فرض می‌کنیم کل برنامه $P_\star$ که متد مورد نظر را در برگرفته است، و نیز گراف فراخوانی جاری که در حال ساخت آن هستیم را در دست داریم. بنابراین، در تعریف‌هایی که ارائه می‌دهیم، $P_\star$ را به عنوان یک پارامتر ضمنی در نظر می‌گیریم.

مجموعه $Classes$ حاوی شناسه‌های تمامی کلاس‌های موجود در برنامه $P_\star$ است. این مجموعه را به‌صورت زیر تعریف

می‌کنیم:

$$Classes = \{c \mid \texttt{class}\ c\ \texttt{extends}\ c'\ \cdots \in P_\star \vee \texttt{class}\ c''\ \texttt{extends}\ c\ \cdots \in P_\star\}$$

ترتیب جزئی زیرکلاس‌بودن به عنوان رابطه‌ای دو موضعی بر روی مجموعه $Classes$ تعریف می‌کنیم، و با نماد $<:$ نشان می‌دهیم. این رابطه را می‌توان با توجه به حکم $P; E \vdash t_۱ <: t_۲$ در سیستم نوع استاندارد، و یا به‌صورت استقرایی زیر تعریف کرد:

$$\forall c \in Classes \cdot c <: c, \tag{۱.۶}$$

$$\forall c_۱, c_۲, c_۳ \in Classes \cdot c_۱ <: c_۲ \wedge\ \texttt{class}\ c_۲\ \texttt{extends}\ c_۳\ \cdots \in P_\star \implies c_۱ <: c_۳. \tag{۲.۶}$$

با توجه به فرض نبود دور در سلسله مراتب کلاس‌ها (در پیوست اول پایان‌نامه) واضح است که $(Classes, <:)$ یک مجموعه جزئاً مرتب است. همچنین، واضح است که (مجموعه $Classes$ و) هر زیرمجموعه از مجموعه $Classes$ در شرایط لم زورن صدق می‌کند.

برای بررسی این موضوع که آیا کلاس c در برنامه $P_\star$ متدی با نام m تعریف کرده است یا خیر، از گزاره زیر استفاده می‌کنیم:

$$decl(c, m) \iff \texttt{class}\ c\ \ldots\ \{\ldots\ m(\ldots)\{\ldots\}\ \ldots\} \in P_\star.$$

عملگر $Maximal$ را به‌صورت زیر تعریف می‌کنیم:

$$Maximal(C) = \{c \mid c \in C \wedge \forall c' \in C \cdot c <: c' \implies c = c'\}$$

به قسمی که $C \subseteq Classes$ است. واضح است که اگر $C \neq \emptyset$ باشد، آنگاه $Maximal(C) \neq \emptyset$ است. همچنین، به راحتی می‌توان مشاهده کرد که اگر C یک زنجیر باشد آنگاه $Maximal(C)$ یک مجموعه تک نقطه‌ای است.

برای ارائه تحلیل‌های سلسه مراتب کلاس‌ها و نیز تحلیل سریع نوع تعریف‌های مقدماتی زیر را ارائه می‌کنیم:

$$HierarchyOf(t) = \{c \mid c <: t\},$$
$$InstantiatedTypes = \{t \mid \texttt{new}\ t \in P_\star\}.$$

با توجه به این دو مجموعه، مجموعه نوع‌های سریع را به‌صورت زیر تعریف می‌کنیم:

$$RapidTypesOf(t) = HierarchyOf(t) \cap InstantiatedTypes.$$

حال به منظور تعریف مجموعه گراف‌های فراخوانی، مجموعه گره‌ها و کمان‌های این مجموعه از گراف‌های جهت‌دار را

به‌صورت زیر تعریف می‌کنیم:

$$Nodes = Methods \times \mathcal{P}(Labels),$$
$$Arcs = Labels \times Nodes.$$

با توجه به این مجموعه‌ها، مجموعه گراف‌های فراخوانی را به‌صورت زیر تعریف می‌کنیم:

$$CallGraphs = \{(N, A) \mid N \in Nodes \wedge A \in Arcs\}.$$

هر گراف فراخوانی $G \in CallGraphs$ که در شرایط زیر صدق کند، یک گراف فراخوانی سازگار می‌نامیم:

$$\forall n_1, n_2 \in N \cdot \pi^1(n_1) = \pi^1(n_2) \implies n_1 = n_2, \tag{۳.۶}$$

$$\forall c \in Classes \cdot \forall m \cdot decl(c, m) \implies \left(\exists n \in N \cdot \pi^1(n) = c.m\right). \tag{۴.۶}$$

که در آن $G = (N, A)$ است.

حال تعدادی تابع کمکی برای دستکاری گراف‌های فراخوانی تعریف می‌کنیم. تابع $NodeOf$ با دریافت یک گراف فراخوانی و نام یک متد، گرهی از آن گراف که مربوط به متد مورد نظر است را بر می‌گرداند:

$$NodeOf(G, c.md) = n \qquad ; G = (N, A) \wedge n \in N \wedge \pi^1(n) = c.md$$

واضح است که $NodeOf$ تابعی جزئی است. اما در این پایان‌نامه در مواقعی که از آن استفاده می‌شود، تابع بر روی تمامی اسامی متدهای موجود در برنامه‌ای که درحال تحلیل آن هستیم، تعریف شده است. چرا که در حین تحلیل هر متد یک گراف فراخوانی به عنوان مقدار اولیه در اختیار داریم که در آن به ازای هر نام متد در برنامه دقیقاً یک گره موجود است.

در صورتی که T زنجیری از شناسه کلاس‌ها باشد، توابع کمکی زیر را تعریف می‌کنیم:

$$TargetMethods(T, m) = \{c.m \mid c \in T \wedge decl(c, m)\} \cup TM(T, m)$$

$$TM(T, m) = \begin{cases} \emptyset & ; c \in Maximal(T) \wedge decl(c, m) \\ \{c'\} & ; c \in Maximal(T) \wedge \neg decl(c, m) \\ & \wedge c <: c' \wedge decl(c', m) \wedge \forall c'' \cdot (c <: c'' \wedge decl(c'', m)) \implies c' <: c''. \end{cases}$$

که در آن $m \in Identifiers$، نام یک متد است.

در نهایت تابع کمکی زیر را تعریف می‌کنیم:

$$TargetNodes(G, M) = \{n \mid G = (N, A) \wedge n \in N \wedge \pi^1(n) \in M\},$$

که در آن G یک گراف فراخوانی، و $M \subseteq Methods$ است.

حال با استفاده از یک تحلیل مبتنی بر محدودیت‌ها گراف فراخوانی برای یک عبارت مانند e را تعریف می‌کنیم. قبل از ارائه یک تحلیل مبتنی بر محدودیت‌ها لازم است در مورد مجموعه گراف‌های فراخوانی بحث کنیم. عملگر $\sqsubseteq$ را به‌صورت زیر بر روی این مجموعه تعریف می‌کنیم:

$$G_1, G_2 \in CallGraphs$$
$$G_1 = (N_1, A_1), G_2 = (N_2, A_2)$$

$$G_1 \sqsubseteq G_2 \iff N_1 \subseteq N_2 \wedge A_1 \subseteq A_2.$$

واضح است که $(CallGraphs, \sqsubseteq)$ یک مشبک تام است. در صورتی که خود را به شناسه‌ها و برچسب‌های موجود در برنامه‌ای که در حال تحلیل آن هستیم محدود کنیم، این مشبک (متناهی) شرط زنجیر صعودی را نیز برآورده خواهد کرد.

در صورتی که G_i، گراف فراخوانی اولیه، یک گراف فراخوانی سازگار باشد. کوچکترین گراف فراخوانی سازگاری که در شرایط $\mathcal{C}_{G_i}[\![e]\!]$ صدق می‌کند، را گراف فراخوانی عبارت e می‌گوییم. مجموعه محدودیت‌های استخراج شده، برای یک عبارت مانند e به‌صورت استقرایی زیر قابل تعریف است.

$$\mathcal{C}_{G_i}[\![[\texttt{new } t]^\ell]\!] = \emptyset \qquad \mathcal{C}_{G_i}[\![[e.fd]^\ell]\!] = \mathcal{C}_{G_i}[\![e]\!]$$
$$\mathcal{C}_{G_i}[\![[var]^\ell]\!] = \emptyset \qquad \mathcal{C}_{G_i}[\![[e_1.fd = e_2]^\ell]\!] = \mathcal{C}_{G_i}[\![e_1]\!] \cup \mathcal{C}_{G_i}[\![e_2]\!]$$
$$\mathcal{C}_{G_i}[\![[\texttt{null}]^\ell]\!] = \emptyset \qquad \mathcal{C}_{G_i}[\![[(t)e]^\ell]\!] = \mathcal{C}_{G_i}[\![e]\!]$$
$$\mathcal{C}_{G_i}[\![[\texttt{let } var = e_1 \texttt{ in } e_2]^\ell]\!] = \mathcal{C}_{G_i}[\![e_1]\!] \cup \mathcal{C}_{G_i}[\![e_2]\!] \qquad \mathcal{C}_{G_i}[\![[\texttt{synchronized } e_1 \texttt{ in } e_2]^\ell]\!] = \mathcal{C}_{G_i}[\![e_1]\!] \cup \mathcal{C}_{G_i}[\![e_2]\!]$$

$$\mathcal{C}_{G_i}[\![[e..md(md(e_1, \ldots, e_k)]^\ell]\!] = (\bigcup\{\mathcal{C}_{G_i}[\![e_i]\!] \mid 0 \le i \le k\})$$
$$\cup\{\{\ell\} \subseteq \pi^2(NodeOf(G_i, c'.md')), \{\ell\} \times TargetNodes(G_i, M) \subseteq A\}.$$

که در آن:

$$M = TargetMethods(RapidTypesOf(t), md),$$
$$e. : t,$$
$$\texttt{class } c' \ \ldots\{\ldots\ md'(\ldots)\{\ldots\ [\ldots]^\ell\ \ldots\}\ \ldots\} \in P_\star,$$
$$G_i = (N, A).$$

در نهایت داریم:

$$\mathcal{C}_{G_i}[\![[e.\texttt{start}()]^\ell]\!] = \mathcal{C}_{G_i}[\![e]\!] \cup \{\{\ell\} \subseteq \pi^2(NodeOf(G_i, c'.md')), \{\ell\} \times TargetNodes(G_i, M) \subseteq A\}.$$

که در آن:

$$M = TargetMethods(RapidTypesOf(t), \texttt{run}),$$
$$e : t,$$
$$\texttt{class}\ c'\ \ldots\{\ldots\ md'(\ldots)\{\ldots\ [\ldots]^{\ell}\ \ldots\}\ \ldots\} \in P_\star,$$
$$G_i = (N, A).$$

توجه کنید که این محدودیت‌ها را می‌توان به کمک یک الگوریتم محاسبه نقطه ثابت (مثلاً یک SAT solver) حل کرد. بررسی چنین الگوریتم‌هایی فراتر از بحث ما در این پایان نامه است.

در صورتی که $CallSites \subseteq Labels$ را به عنوان مجموعه محل‌های فراخوانی تعریف کنیم، تابع $callees$ برای یک محل فراخوانی، مجموعه اسامی متدهایی را می‌دهد که ممکن است در زمان اجرا فراخوانی شوند. این تابع را با فرض وجود یک گراف فراخوانی $G = (N, A)$ برای عبارت e که $\ell \in labels(e)$، به‌صورت زیر تعریف می‌کنیم:

$$callees : CallSites \to \mathcal{P}(Methods)$$

$$callees(\ell) = \{n \mid (\ell, n) \in A\}.$$

تابع $mayCall$ با دریافت نام کامل یک متد، مجموعه اسامی تمامی متدهایی را برمی‌گرداند که در زمان اجرا ممکن است توسط آن متد فراخوانی شود:

$$mayCall : Methods \to \mathcal{P}(Methods)$$

$$mayCall(m) = \bigcup\{callees(\ell) \mid \ell \in \pi^{۲}(NodeOf(m))\}.$$

۲.۶ تحلیل اشاره‌گر

در زبان مورد بررسی در این پایان‌نامه هر عبارت در نهایت به یک شئ ارزیابی می‌شود. به عبارت دیگر، هر عبارت مانند یک اشاره‌گر یا ارجاع است که به یک شئ اشاره دارد. هدف از یک تحلیل اشاره‌گر ساخت تابعی است که هر عبارت یک برنامه (که با برچسب‌های منحصر به فرد قابل شناسایی هستند) را به مقادیری که به نحوی نشان دهنده مقصد ارجاع آن عبارت است، نگاشت می‌کند. به‌طور معمول، چنین تابعی از روی گراف‌های اشاره‌گر (یا در برخی از موارد از روی گراف‌های شکل هیپ) قابل تولید است. بنابراین، تابع یاد شده هر برچسب عبارت را به یک گره گراف اشاره‌گر (یا گراف شکل هیپ) نگاشت می‌کند.

در این بخش شیوه استخراج یک گراف اشاره‌گر از روی متن برنامه‌ها را شرح می‌دهیم. برای این منظور لازم است چند تعریف مربوط به گراف اشاره‌گر و شکل هیپ را ارئه دهیم.

تعریف ۱.۶. مجموعه گره‌های داخلی N_I، به گره‌هایی گفته می‌شود که در یک تحلیل اشاره‌گر نشان دهنده مجموعه

شئ‌هایی است که توسط یک عملگر نمونه‌سازی نوع (new) ایجاد می‌شوند. این مجموعه را می‌توان به‌صورت زیر تعریف کرد:

$$N_I = \{\mathtt{i}(id, m) \mid id \in Labels^+ \wedge m \in Multiplicities\}$$

همان‌طور که مشاهده می‌شود، هر گره حاوی اطلاعات چندگانگی m، و یک شناسه است. مجموعه گره‌های خارجی N_E، نشان دهنده مجموعه تمامی شئ‌هایی است که متد مورد تحلیل انتظار دارد یک متن فراخوانی فراهم کند. این مجموعه را به‌صورت زیر تعریف می‌کنیم:

$$N_E = \{\mathtt{e}(n) \mid n \in \mathbb{N}\}$$

از آنجایی که چندگانگی گره‌های خارجی همواره نامشخص «?» است، این گره‌ها حاوی اطلاعات صریح چندگانگی نیست. مجموعه گره‌های برگشتی N_R، شامل تمامی گره‌های خارجی یا داخلی است که این گره‌ها نشان دهنده شئ‌هایی است که انتظار می‌رود متد مورد بررسی به عنوان مقدار برگشتی، برگشت دهد. این مجموعه را به‌صورت زیر تعریف می‌کنیم:

$$N_R = \{\mathtt{r}(n) \mid n \in (N_E + N_I)\}$$

در نهایت، مجموعه تمامی گره‌ها را به‌صورت زیر تعریف می‌کنیم:

$$N = N_I + N_E + N_R$$

(توجه: در این تعریف «+» نماد اجتماع منفصل است.) ■

برای بررسی نوع یک گره تعریف‌های کمکی زیر را ارائه می‌دهیم:

$$\begin{aligned} &n \in N : \\ &\quad int(n) \iff n \in N_I \\ &\quad ext(n) \iff n \in N_E \\ &\quad ret(n) \iff n \in N_R \end{aligned}$$

برای ساخت یک گره از یک نوع به‌خصوص نیز تعریف‌های زیر را ارائه می‌دهیم:

$$\begin{aligned} mkInt(id, m) &= \mathtt{i}(id, m) && id \in Labels^+ \\ & && , m \in Multiplicities \\ mkExt(n) &= \mathtt{e}(n) && n \in \mathbb{N} \\ mkRet(n) &= \mathtt{r}(n) && n \in (N_E + N_I) \end{aligned}$$

تعریف ۲.۶. مجموعه مقادیر مجرد، به‌طور معمول به مجموعه‌ای از مقادیر گفته می‌شود که یک تحلیل ایستا، به عنوان تخمینی از مقدار نهایی یک عبارت (در زمان اجرا) محاسبه می‌کند. برای تحلیل اشاره‌گر ارائه شده در این پایان‌نامه مجموعه

مقادیر مجرد را به‌صورت زیر تعریف می‌کنیم:

$$Val = \mathcal{P}(N)$$

■

با توجه به تعریف مقدار مجرد، چند مفهوم دیگر که برای توسعه تحلیل نیاز داریم، قابل تعریف است.

تعریف ۳.۶. مخزن مجرد، به‌طور معمول به هر نگاشت بین عبارت‌های یک برنامه و مقادیر مجرد گفته می‌شود. از آنجایی که در زبان مورد بررسی در این پایان‌نامه عبارات به‌صورت منحصر به فرد با استفاده از برچسب‌ها قابل شناسایی هستند، این نگاشت را به‌صورت زیر تعریف می‌کنیم:

$$Cache = \mathcal{P}(Labels \times Val)$$

واضح است که این مجموعه با توجه به $\supseteq$، $\bigcup$، و $\bot_C = \emptyset$ یک مشبک تام است. ■

تعریف ۴.۶. حالت مجرد به‌طور معمول به هر نگاشت بین متغیرهای یک برنامه و مقادیر مجرد گفته می‌شود. این نگاشت را به‌صورت زیر تعریف می‌کنیم:

$$State = \mathcal{P}(Var \times Val)$$

واضح است که این مجموعه با توجه به $\supseteq$، $\bigcup$، و $\bot_S = \emptyset$ یک مشبک تام است. ■

تعریف ۵.۶. یک هیپ مجرد برای یک برنامه پاسخ یک تحلیل شکل هیپ است که در اصل تخمینی از هیپ در زمان اجرا برای آن برنامه است. ما هیپ مجرد را به‌صورت زیر تعریف می‌کنیم:

$$Heap = \{(ALoc, ARef_I, ARef_E) \mid ALoc \subseteq N \wedge ARef_I, ARef_E \subseteq ALoc \times ALoc\}$$

مجموعه $ARef_I$ را مجموعه یال‌های داخلی، و مجموعه $ARef_E$ را مجموعه یال‌های خارجی یک گراف شکل هیپ می‌گوییم. عملگر $\sqsubseteq_H$ بر روی مجموعه $Heap$ به‌صورت زیر قابل تعریف است:

$$h_1, h_2 \in Heap$$
$$h_1 = (ALoc_1, ARef_{I_1}, ARef_{E_1}), h_2 = (ALoc_2, ARef_{I_2}, ARef_{E_2})$$

$$h_1 \sqsubseteq_H h_2 \iff ALoc_1 \subseteq ALoc_2 \wedge ARef_{I_1} \subseteq ARef_{I_2} \wedge ARef_{E_1} \subseteq ARef_{E_2}$$

در صورتی که $H \subseteq Heap$ یک زیرمجموعه دلخواه از هیپ‌های مجرد باشد، عملگر کوچکترین کران بالا برای این مجموعه را به‌صورت زیر تعریف می‌کنیم:

$$\bigsqcup_H H = (\bigcup ALoc', \bigcup ARef'_I, \bigcup ARef'_E)$$

که در آن $ALoc' = \{ALoc \mid (ALoc, ARef_I, ARef_E) \in H\}$، $ARef'_I = \{ARef_I \mid (ALoc, ARef_I, ARef_E) \in H\}$، و $ARef'_E = \{ARef_E \mid (ALoc, ARef_I, ARef_E) \in H\}$ است. واضح است که این مجموعه با توجه به وجود عملگرهای $\sqsubseteq_H$ و $\bigsqcup_H$ یک مشبک تام است. کوچکترین عنصر این مشبک $\bot_H = (\emptyset, \emptyset, \emptyset)$ است. ■

در نهایت دو مفهوم اساسی برای ارائه تحلیل اشاره‌گر لازم است که در تعریف زیر معرفی می‌کنیم.

تعریف ۶.۶. مجموعه تمام گراف‌های اشاره‌گر را به‌صورت زیر تعریف می‌کنیم:

$$PG = \{(S, H) \mid S \in State \wedge H \in Heap\}$$

واضح است که این مجموعه با توجه به عملگرها زیر، و مقدار $\bot_\pi = (\bot_S, \bot_H)$ یک مشبک تام است.

$$\pi_1, \pi_2 \in PG$$
$$\pi_1 = (S_1, H_1), \pi_2 = (S_2, H_2)$$
$$\pi_1 \sqsubseteq_\pi \pi_2 \iff S_1 \subseteq S_2 \wedge H_1 \sqsubseteq_H H_2$$

$$P \subseteq PG$$
$$\bigsqcup_\pi P = (\bigcup S', \bigsqcup_H H')$$

که در آن $S' = \{S \mid (S, H) \in P\}$ و $H' = \{H \mid (S, H) \in P\}$.

مجموعه اطلاعات اشاره‌گر را به‌صورت زیر تعریف می‌کنیم:

$$PI = \{(C, \pi) \mid C \in Cache \wedge \pi \in PG\}$$

واضح است که این مجموعه با توجه به عملگرهای زیر، و مقدار $\bot_P = (\bot_C, \bot_\pi)$ یک مشبک تام است.

$$p_1, p_2 \in PI$$
$$p_1 = (C_1, \pi_1), p_2 = (C_2, \pi_2)$$
$$p_1 \sqsubseteq_P p_2 \iff C_1 \subseteq C_2 \wedge \pi_1 \sqsubseteq_\pi \pi_2$$

$$P \subseteq PI$$
$$\bigsqcup_P P = (\bigcup C', \bigsqcup_\pi P')$$

که در آن $C' = \{C \mid (C, \pi) \in P\}$ و $P' = \{\pi \mid (C, \pi) \in P\}$. ■

۱.۲.۶ مجموعه‌های دگرنامی

مفهوم مجموعه‌های دگرنامی در این پایان‌نامه، در داخل متدها برای کسب اطلاعات بیشتر از متن‌های فراخوانی مورد استفاده قرار می‌گیرد. برای این منظور نحو زبان جاوای مورد بررسی را به‌صورت زیر گسترش می‌دهیم:

$meth$	::=	$asdec^{?}\ t\ md(par^{*})\ asa^{?}\ \{\ e\ \}$	تعریف متد
$asdec$	::=	`aset` $asid^{*}$	اعلان شناسه مجموعه دگرنامی
par	::=	$t\ var\ asa^{?}$	پارامتر
asa	::=	`in` $asid$	انتساب شناسه مجموعه دگرنامی
$asid$	$\in$	$Identifiers$	شناسه‌های مجموعه‌های دگرنامی

شکل ۱.۶: متغیرهای نحوی گسترش یافته

به‌طور پیش‌فرض هر پارامتر در یک مجموعه دگرنامی منحصر به فرد قرار می‌گیرد. بنابراین، نوشتن حاشیه‌نوشته‌های مربوط به مجموعه‌های دگرنامی در اعلان متدها اختیاری است (این موضوع در نحو مجرد زبان در جدول بالا با استفاده از علامت ? منعکس شده است.) حاشیه‌نوشته `aset` برای اعلان تعدادی متناهی شناسه مجموعه دگرنامی مورد استفاده قرار می‌گیرد. حوزه تعریف هر یک از شناسه‌ها، صورت متد (به استثنای بدنه آن) است. مجموعه دگرنامی پیش‌فرض هر یک از پارامترهای متد مورد نظر را می‌توان با حاشیه‌نوشته `in` n به n تغییر داد. حاشیه‌نوشته‌های مربوط به پارامتر ویژه `this`، بعد از لیست پارامترهای متد و قبل از شروع بدنه آن قرار دارد. بدیهی است که متن‌های فراخوانی نیز باید از این شیوه قرار گرفتن آرگومان‌ها (به تبعیت از شیوه قرار گرفتن پارامترها) در مجموعه‌های دگرنامی پیروی کند. در بخش پایانی این بخش با ارائه یک سیستم نوع این موضوع را صحت سنجی می‌کنیم.

در تحلیل ارائه شده در این فصل پیمانه‌ای بودن را، همانند تحلیل فصل چهارم، با استفاده از استخراج خلاصه‌های متدها فراهم کرده‌ایم. برخلاف اطلاعات مورد نیاز در داخل هر متد، اطلاعاتی که در بیرون از متدها نیاز داریم (یعنی خلاصه متد) بدون ادغام پارامترها با استفاده از مجموعه‌های دگرنامی، دقیق‌تر هستند. بنابراین، برای تحلیل بدنه متدها به‌صورت مستقیم از مشبک $(PI, \sqsubseteq_P, \bigsqcup_P)$ استفاده نخواهیم کرد. بلکه از یک مشبک که اعضای آن زوج‌های مرتبی از مجموعه PI است استفاده می‌کنیم:

$$PI^{\star} = PI \times PI$$

$$p_1, p_2 \in PI^{\star}$$
$$p_1 = (p_{11}, p_{12}), p_2 = (p_{21}, p_{22})$$
$$p_1 \sqsubseteq_{\star} p_2 \iff p_{11} \sqsubseteq_P p_{21} \wedge p_{12} \sqsubseteq_P p_{22}$$

$$P \subseteq PI^{\star}:$$
$$\bigsqcup_{\star} P = (\bigsqcup_P P'_1, \bigsqcup_P P'_2)$$

که در آن $P'_1 = \{P_1 \mid (P_1, P_2) \in P\}$ و $P'_2 = \{P_2 \mid (P_1, P_2) \in P\}$. واضح است که $(PI^{\star}, \sqsubseteq_{\star}, \bigsqcup_{\star})$ یک مشبک تام است. مؤلفه اول یک عضو مجموعه $PI^{\star}$ را اطلاعات اشاره‌گری خارجی و مؤلفه دوم آن را اطلاعات اشاره‌گری داخلی گوییم.

تنها تفاوت این دو مقدار در شیوه قرار گرفتن پارامترهای متدها در مجموعه‌های دگرنامی، و در نتیجه تنها در مقدار اولیه فراهم شده برای تحلیل اشاره‌گر است. در ادامه به‌طور دقیق‌تر به این موضوع خواهیم پرداخت، فعلاً کافی است بدانیم که مقدار اولیه فراهم شده برای ساخت مؤلفه اول مستقل از حاشیه‌نوشته‌های دگرنامی بوده، حال آنکه مقدار اولیه لازم برای ساخت مؤلفه دوم بر اساس حاشیه‌نوشته‌های دگرنامی فراهم شده توسط برنامه‌نویس است.

۲.۲.۶ تحلیل جریان داده

تحلیل اشاره‌گر ($PointsTo$) ارائه شده در این پایان‌نامه نمونه‌ای از چارچوب یکنوا است: یک تحلیل مبتنی بر جریان داده، رو به جلو، و با قطعیت may. بنابراین، در حالت کلی تحلیل را می‌توان به‌صورت زیر نوشت:

$$PointsTo_\circ(\ell) = \begin{cases} \iota & ;\ell = init(e) \\ \bigsqcup_\star \{PointsTo_\bullet(\ell') \mid (\ell', \ell) \in flows(e)\} & ;o.w. \end{cases}$$

$$PointsTo_\bullet(\ell) = f_\ell(cs, ms)(PointsTo_\circ(\ell))$$

که در آن e عبارتی است که قصد تحلیل اشاره‌گری آن را داریم، این عبارت می‌تواند عبارت آغازین برنامه یا صورت کامل متدی باشد که در حال تحلیل آن هستیم. مقدار ι، مقدار اولیه تحلیل نام دارد، و همان‌طور که می‌دانیم این مقدار یک دوتایی به‌صورت $\iota = (\iota_۱, \iota_۲)$ است که مؤلفه دوم آن وابسته به صورت متدی است که عبارت e بدنه آن را تشکیل می‌دهد. همچنین، فرض می‌کنیم پارامتر ویژه `this`، پارامتر اول (یعنی پارامتر صفرام) متد مورد نظر، مثلاً متد $meth$ است. حال فرض می‌کنیم $ASID$، مجموعه تمام شناسه‌های ممکن برای مجموعه‌های دگرنامی باشد. بر اساس این فرضیات، مجموعه Par را به عنوان تمام زوج‌های (p, n) تعریف می‌کنیم که در آن p یک شناسه پارامتر اعلان شده برای متد $meth$ به همراه یک حاشیه‌نوشته n `in` n، یا p یک پارامتر اعلان شده برای متد $meth$ بدون حاشیه‌نوشته n `in` بوده و $n \in ASID$ یک شناسه تازه در مجموعه Par است. منظور ما از یک شناسه تازه، یک شناسه مجموعه دگرنامی است که به عنوان مؤلفه دوم مجموعه Par ظاهر نشده باشد. حال تابع h را به عنوان یک تابع یک به یک و کامل از مجموعه $ASID$ به مجموعه گره‌های خارجی (N_E) فرض می‌کنیم. در نهایت $\iota_۱$، و $\iota_۲$ را به‌صورت زیر تعریف می‌کنیم:

$$\iota_۱ = (\bot_C, (g, \bot_H)),$$
$$\iota_۲ = (\bot_C, (h \circ Par, \bot_H)),$$

که در آن g تابعی یک به یک و کامل از مجموعه شناسه‌های پارامترهای اعلان شده در متد $meth$ به مجموعه N_E است. با توجه به این تعریف‌ها، بدیهی است که در حالتی هم که درگیر تحلیل عبارت آغازین برنامه هستیم (یعنی هیچ متدی موجود نیست که عبارت e بدنه آن را تشکیل دهد)، داریم: $\iota = \bot_\star$. لازم به یادآوری است که در حالتی که عبارت e زیرعبارت محضی از عبارت آغازین برنامه یا بدنه یک متد باشد، مقادیر اولیه $\iota_۱$ و $\iota_۲$ نیز باید حاوی اطلاعاتی مناسبی از متغیرهای آزاد موجود در این عبارت باشد. بنابراین، به دست آوردن مقدار اولیه تحلیل تابعی از عبارتی است که در حال تحلیل آن هستیم.

تابع f_ℓ نیز در این تعریف تابع انتقال است که با مجموعه فراخوانی cs، و نگاشت خلاصه متدها ms، پارامتریزه شده است. مجموعه فراخوانی $cs \subseteq Methods$، شامل نام کامل متدهایی است که تحلیل اشاره‌گری برای آن‌ها فراخوانی شده،

اما هنوز پاسخی برای آن‌ها وجود ندارد. در ادامه به تعریف نگاشت خلاصه متدها و نیز تابع انتقال می‌پردازیم. در حالت کلی، تابع انتقال را به‌صورت زیر تعریف می‌شود:

$$f_\ell(cs, ms)(p) = (p - kill([B]^\ell)(cs, ms)(p)) \sqcup_\star gen([B]^\ell)(cs, ms)(p)$$

که در آن $[B]^\ell \in blocks(e)$ است. دقت کنید که عملگر تفاضل زوج‌های اطلاعات اشاره‌گری، به راحتی و با گسترش عملگر تفاضل مجموعه‌ها قابل تعریف است:

$$\begin{aligned} &p_۱, p_۲ \in PI^\star : \\ &\quad p_۱ = (p_{۱۱}, p_{۱۲}), p_۲ = (p_{۲۱}, p_{۲۲}) \\ &\quad p_۱ - p_۲ = (p_{۱۱} - p_{۲۱}, p_{۱۲} - p_{۲۲}) \end{aligned}$$

عملگر تفاصل اطلاعات اشاره‌گری تنها نیز به این ترتیب قابل تعریف است:

$$\begin{aligned} &p_۱, p_۲ \in PI : \\ &\quad p_۱ = (C_۱, \pi_۱), p_۲ = (C_۲, \pi_۲) \\ &\quad p_۱ - p_۲ = (C_۱ - C_۲, \pi_۱ - \pi_۲) \end{aligned}$$

عملگر تفاضل گراف‌های اشاره‌گر را به‌صورت زیر تعریف کنیم:

$$\begin{aligned} &\pi_۱, \pi_۲ \in PG : \\ &\quad \pi_۱ = (S_۱, H_۱), \pi_۲ = (S_۲, H_۲) \\ &\quad \pi_۱ - \pi_۲ = (S_۱ - S_۲, H_۱ - H_۲) \end{aligned}$$

در نهایت عملگر تفاضل برای هیپ‌های مجرد را به‌صورت زیر تعریف می‌کنیم:

$$\begin{aligned} &h_۱, h_۲ \in Heap : \\ &\quad h_۱ = (ALoc_۱, ARef_{I_۱}, ARef_{E_۱}), h_۲ = (ALoc_۲, ARef_{I_۲}, ARef_{E_۲}) \\ &\quad h_۱ - h_۲ = (ALoc_۱ - ALoc_۲, ARef_{I_۱} - ARef_{I_۲}, ARef_{E_۱} - ARef_{E_۲}). \end{aligned}$$

تعریف ۷.۶ (پاسخ تحلیل). اگر e یک عبارت یا صورت کامل یک متد باشد، و ι مقدار اولیه تحلیل اشاره‌گر برای آن باشد، با فرض نگاشت خلاصه متدهای ms، و مجموعه فراخوانی cs به‌طوری که به ازای هر نام متد $c.md$ که $bodyOf(c.md) = e$ داشته باشیم $c.md \in cs$، پاسخ تحلیل اشاره‌گری برای e یک سه‌تایی به‌صورت زیر است:

$$(PointsTo_\circ, PointsTo_\bullet, \mu),$$

که در آن μ به ازای هر مکان فراخوانی، نگاشتی بین گره‌های خارجی موجود در خلاصه متد(های) فراخوانی شده، و گره‌های

گراف هیپ مجرد e است. تعریف دقیق این نگاشت، و شیوه به دست آوردن آن در ادامه این بخش به‌طور دقیق شرح داده شده است. ■

از آنجایی که عبارات زبان جاوای مورد بررسی در این پایان نامه تأثیر یکسانی بر روی هر دو مؤلفه اطلاعات اشاره‌گری دارد، توابع gen و $kill$ را صرفاً برای یکی از آن‌ها و بر روی مجموعه PI تعریف می‌کنیم. همچنین، برای سادگی در نمایش در حین تعریف توابع انتقال، از نوشتن پارامترهای cs و ms در مواقعی که توابع به آن‌ها وابسته نیستند خودداری خواهیم کرد. در ادامه این فصل اثر هر یک از عبارات را تک تک مورد بررسی قرار می‌دهیم.

تابع انتقال برای $[var]^\ell$: ارزیابی چنین عبارتی موجب بروزرسانی مخزن مجرد به‌صورتی می‌شود که در آن برچسب ℓ به مقدار متغیر در حالت جاری اشاره کند. بنابراین، توابع $kill$ و gen را به‌صورت زیر تعریف می‌کنیم:

$$kill([var]^\ell)(p) = \bot_P$$

$$gen([var]^\ell)(p) = \begin{cases} \bot_P & ; var \notin dom(S) \\ (\{\ell\} \times S(var), \bot_\pi) & ; o.w. \end{cases}$$

که در آن $p = (C, (S, H))$ اطلاعات اشاره‌گری جاری است.

تابع انتقال برای $[\texttt{null}]^\ell$: از آنجایی که ارزیابی چنین عبارتی هیچ تأثیری بر روی اطلاعات جاری اشاره‌گری ندارد، توابع $kill$ و gen نیز به به‌صورت ساده زیر تعریف می‌شوند:

$$kill([\texttt{null}]^\ell) = \bot_P$$

$$gen([\texttt{null}]^\ell) = \bot_P$$

تابع انتقال برای $[e.fd]^\ell$: تابع $kill$ برای چنین عبارت‌هایی هیچ کاری انجام نمی‌دهد، بنابراین آن را به‌صورت زیر تعریف می‌کنیم:

$$kill([e.fd]^\ell)(p) = \bot_P$$

اما با فرض $p = (C, (S, H))$، ممکن است همه مؤلفه‌ها به غیر از S تحت تأثیر قرار گیرند. بنابراین، اگر فرض کنیم

$$N' = \bigcup \{C(\ell') \mid \ell' \in final(e) \text{ تعریف شده است } C \text{ بر روی}\}$$

تابع gen را به‌صورت زیر تعریف می‌کنیم:

$$gen([e.fd]^\ell)(p) = \begin{cases} \bot_P & ; N' = \emptyset \\ \bigsqcup_P \{g(n, fd) \mid n \in N'\} & ; o.w. \end{cases}$$

که در آن g را به‌صورت زیر تعریف می‌کنیم:

$$g(n, fd) = \begin{cases} \bot_P & ; \quad int(n) \text{ و } H \text{ بر روی} \\ & \quad (n, fd) \text{ تعریف نشده باشد} \\ (\{\ell\} \times N'', \bot_\pi) & ; \quad \text{مجموعه ناتهی } N'' \subseteq N \text{ موجود باشد به قسمی که،} \\ & \quad \text{به ازای هر } n' \in N'' \text{ داشته باشیم } (n, fd, n') \in H \\ (\{(\ell, n)\}, (\bot_S, (\{n'\}, \emptyset, \{(n, fd, n')\}))) & ; \quad ext(n) \text{ و } H \text{ بر روی } (n, fd) \\ & \quad \text{تعریف نشده، و } n' \text{ در } H \text{ تازه است، و } ext(n') \end{cases}$$

بند آخر تابع تنها قسمی است که لازم است قدری توضیح دهیم: در این بند به حالتی رسیدگی می‌شود که در آن بدنه متد از یک گره خارجی انتظار دارد با فیلد fd به جایی شیءای اشاره کند. در این صورت تابع با ایجاد گره‌ای خارجی و با اتصال گره مورد نظر به گره تازه ایجاد شده توسط یک یال خارجی این انتظار را توصیف می‌کند. در نهایت لازم به یاد آوری است که گره تازه خارجی را می‌توان با استفاده از یک عدد طبیعی و تابع کمکی $mkExt$ ساخت.

تابع انتقال برای $[\texttt{new } t]^\ell$: با فرض $p = (C, (S, H))$، این دسته از عبارات تنها در C و H تأثیر می‌گذارند. با این حساب توابع $kill$ و gen را به‌صورت زیر تعریف می‌کنیم:

$$kill([\texttt{new } t]^\ell)(p) = \bot_P$$

$$gen([\texttt{new } t]^\ell)(p) = (\{(\ell, n)\}, (\bot_S, (\{n\}, \emptyset, \emptyset)))$$

که در آن $n = mkInt(\ell, getLabMult(\ell))$ یک گره داخلی تازه در H است.

تابع انتقال برای $[e_1.fd = e_2]^\ell$: با فرض $p = (C, (S, H))$، دو مجموعه زیر را تعریف می‌کنیم:

$$N_1' = \bigcup\{C(\ell') \mid \ell' \in final(e_1) \text{ تعریف شده است } | \ C \text{ بر روی}\}$$

$$N_2' = \bigcup\{C(\ell') \mid \ell' \in final(e_2) \text{ تعریف شده است } | \ C \text{ بر روی}\}$$

بر اساس این تعریف‌ها و فرضیات، تابع‌های $kill$ و gen را به‌صورت زیر تعریف می‌کنیم:

$$kill([e_1.fd = e_2]^\ell)(p) = \bot_P$$

$$gen([e_1.fd = e_2]^\ell)(p) = \begin{cases} \bot_P & ; N'_1 = \emptyset \\ (\{\ell\} \times N'_2, (\bot_S, (\emptyset, N'_1 \times \{fd\} \times N'_2, \emptyset))) & ; o.w. \end{cases}$$

تابع انتقال برای $[(t)\ e]^\ell$: با فرض $p = (C, (S, H))$، مجموعه N' را به‌صورت زیر تعریف می‌کنیم:

$$N' = \bigcup \{C(\ell') \mid \ell' \in final(e) \text{ تعریف شده است } C \text{ بر روی}\}$$

با این حساب، مخزن مجرد جاری به‌صورت زیر بروزرسانی می‌شود:

$$kill([(t)\ e]^\ell)(p) = \bot_P$$

$$gen([(t)\ e]^\ell)(p) = (\{\ell\} \times N', \bot_\pi).$$

تابع انتقال برای $[begin(var = e_1; e_2)]^{\ell_n}$: با فرض $p = (C, (S, H))$، مجموعه N' را به‌صورت زیر تعریف می‌کنیم:

$$N' = \bigcup \{C(\ell') \mid \ell' \in final(e_1) \text{ تعریف شده است } C \text{ بر روی}\}$$

با این حساب، حالت مجرد جاری به‌صورت زیر بروزرسانی می‌شود:

$$kill([begin(var = e_1; e_2)]^{\ell_n})(p) = \bot_P$$

$$gen([begin(var = e_1; e_2)]^{\ell_n})(p) = (\bot_C, (\{var\} \times N', \bot_H))$$

تابع انتقال برای $[end(var = e_1; e_2)]^{\ell_x}$: با فرض $p = (C, (S, H))$، مجموعه‌های S' و N' را به‌صورت زیر تعریف می‌کنیم:

$$S' = \{(var, n) \mid n \in N \text{ و } (var, n) \in S\}$$

$$N' = \bigcup \{C(\ell') \mid \ell' \in final(e_2) \text{ تعریف شده است } C \text{ بر روی}\}$$

با این فرضیات، توابع $kill$ و gen را به‌صورت زیر تعریف می‌کنیم:

$$kill([end(var = e_1; e_2)]^{\ell_x})(p) = (\bot_C, (S', \bot_H))$$

$$gen([end(var = e_1; e_2)]^{\ell_x})(p) = (\{\ell_x\} \times N', \bot_\pi)$$

تابع انتقال برای $[enterMonitor(e_1; e_2)]^{\ell_n}$: تابع انتقال برای این دسته از بلوک‌ها کار مؤثری را انجام نمی‌دهند:

$$kill([enterMonitor(e_1; e_2)]^{\ell_n}) = \bot_P$$

$$gen([enterMonitor(e_1; e_2)]^{\ell_n}) = \bot_P$$

تابع انتقال برای $[exitMonitor(e_1; e_2)]^{\ell_x}$: با فرض $p = (C, (S, H))$، مجموعه N' را به‌صورت زیر تعریف می‌کنیم:

$$N' = \bigcup \{C(\ell') \mid \ell' \in final(e_2) \text{ بر روی } C \text{ تعریف شده است}\}$$

با این فرضیات، توابع $kill$ و gen را به‌صورت زیر تعریف می‌کنیم:

$$kill([exitMonitor(var = e_1; e_2)]^{\ell_x})(p) = \bot_P$$

$$gen([exitMonitor(var = e_1; e_2)]^{\ell_x})(p) = (\{\ell_x\} \times N', \bot_\pi)$$

تابع انتقال برای $[e..md(e_1, \ldots, e_k)]^{\ell}$: تعریف این تابع انتقال از جمله مواقعی است که به پارامترهای cs و ms نیاز پیدا می‌کنیم. پارامتر ms یا همان نگاشت خلاصه متدها، نگاشتی از نام کامل متدها به مجموعه گراف‌های اشاره‌گر است، که در ادامه این بخش با تعریف دقیق این نگاشت بیشتر آشنا می‌شویم. پارامتر cs شامل مجموعه اسامی متدهایی است که تحلیل اشاره‌گری را برای تحلیل آن‌ها فراخوانی کرده‌ایم و هنوز جوابی برای آن‌ها بدست نیاورده‌ایم.

تابع $kill$ برای این دسته از عبارات کار جالب توجهی انجام نمی‌دهد. این تابع را به‌صورت زیر تعریف می‌کنیم:

$$kill([e..md(e_1, \ldots, e_k)]^{\ell})(cs, ms)(p) = \bot_P$$

طبق معمول، پارامتر ویژه `this` را به عنوان پارامتر صفرام متدها در نظر می‌گیریم. بر این اساس، و با فرض $p =$

$(C,(S,H))$، مجموعه‌های $N'_{e.}$، N'_{e_1}، تا N'_{e_k} را به‌صورت زیر تعریف می‌کنیم:

$$N'_{e.} = \bigcup\{C(\ell') \mid \ell' \in final(e.) \text{ تعریف شده است } C \text{ بر روی}\}$$

$$N'_{e_1} = \bigcup\{C(\ell') \mid \ell' \in final(e_1) \text{ تعریف شده است } C \text{ بر روی}\}$$

$$\vdots$$

$$N'_{e_k} = \bigcup\{C(\ell') \mid \ell' \in final(e_k) \text{ تعریف شده است } C \text{ بر روی}\}$$

همان‌طور که مشاهده می‌شود، N'_{e_i} مجموعه تمام گره‌هایی است که عبارت e_i ممکن است به آن اشاره کند ($0 \le i \le k$.)
صورت کلی تابع gen به‌صورت زیر تعریف می‌شود:

$$gen([e..md(e_1,\ldots,e_k)]^{\ell})(cs,ms)(p) = \bigsqcup_P\{map(N'_{e.},\ldots,N'_{e_k},ms(m),\ell,p) \mid m \in callees(\ell) \wedge m \notin cs\}$$

تابع map برای نگاشت گراف اشاره‌گر حاصل شده از خلاصه متدها بر روی گراف اشاره‌گر جاری، و نیز بروز رسانی مخزن مجرد جاری به‌کار می‌رود. این تابع همچنین نگاشت μ که در تحلیل رخدادها مورد استفاده قرارمی‌گیرد را می‌سازد. برای تعریف تابع map لازم است بدانیم، که این دسته از عبارت‌ها حالت مجرد جاری را تغییر نمی‌دهند حال آنکه مخرن و هیپ مجرد ممکن است تحت تأثیر قرار گیرد. در این قسمت از پایان‌نامه الگوریتمی برای محاسبه تابع map ارائه داده‌ایم. این الگوریتم را بر حسب چند رویه نوشته‌ایم که در ادامه هر یک از این رویه‌ها را شرح می‌دهیم. قبل از تشریح الگوریتم، لازم است یادآوری کنیم که انتساب چندتایی‌ها در اصل به‌صورت مؤلفه به مؤلفه صورت می‌گیرد.
که در آن مجموعه $N' = \{n \mid \mathtt{r}(n) \in (N_R \cap ALoc')\}$ در نظر گرفته شده است. علاوه بر آن فرض کرده‌ایم که p_i شناسه پارامتر iام متد فراخوانی شده است (به‌طور خاص، $p. \triangleq \mathtt{this}$.) لازم به یادآوری است که استخراج نام پارامترها از روی خلاصه متد امکان پذیر است. در نهایت لازم به یادآوری است که $\widehat{S}(p_i)$ برای پارامتر p_i همواره برابر با یک مجموعه تک عضوی است. دستور **return** نیز برای خروج از رویه به‌کار رفته است. حال رویه WorkHorse را در شکل ۳.۶ تعریف می‌کنیم. همان‌طور که مشاهده می‌شود، این رویه نگاشت μ را بروز رسانی می‌کند. این نگاشت که به‌صورت زیر قابل تعریف است:

$$\mu : Labels \to \mathcal{P}(N_E \times (N_I + N_E)),$$

همراه جواب تحلیل اشاره‌گری به مرحله تحلیل رخدادها ارسال می‌شود.
که در آن رویه Remove برای حذف یک گره از چندتایی p' در الگوریتم شکل ۲.۶ به‌کار می‌رود. این رویه در شکل ۴.۶ فهرست شده است.

رویه CleanRetNodes برای تغییر نوع تمامی گره‌های برگشتی به گره‌های معمولی به‌کار می‌رود. این رویه را در شکل ۵.۶ تعریف کرده‌ایم.

رویه Instantiate برای تعیین چندگانگی و نیز تعیین شناسه گره‌های داخلی موجود در خلاصه متدها، در مکان‌های

الگوریتم: محاسبه تابع map.

ورودی: $N'_{e.}, \ldots, N'_{e_k}$، خلاصه متد فراخوانی شده s که یا تهی است یا برابر با مجموعه تک نقطه‌ای $\{(\widehat{S}, (\widehat{ALoc}, \widehat{ARef_I}, \widehat{ARef_E}))\}$ است، برچسب عبارت جاری ℓ، و اطلاعات اشاره‌گری جاری $(C, (S, (ALoc, ARef_I, ARef_E)))$.

خروجی: $p' = (C', (\bot_S, (ALoc', ARef'_I, ARef'_E)))$

1: **if** $s = \emptyset$ **then**
2: $p' := \bot_P$; $C' := \{\ell\} \times \bigcup\{N'_{e_j} \mid \cdot \leq j \leq k\}$; **return**;
3: $(ALoc', ARef'_I, ARef'_E) := (\widehat{ALoc}, \widehat{ARef_I}, \widehat{ARef_E})$;
4: INSTANTIATE(ℓ);
5: **for** $i := 0$ **to** k **do**
6: **let** $\{n\} = \widehat{S}(p_i)$ **in** WORKHORSE(N'_{e_i}, n);
7: $C' := \{\ell\} \times N'$;
8: CLEANRETNODES();

شکل ۲.۶: الگوریتم MAP

WORKHORSE(N,n)$\triangleq$

1: **if** $N = \emptyset$ **then return**;
2: **for each** $n' \in N$ **do**
3: $\mu(\ell) := \mu(\ell) \cup (n', n)$; UNIFY($n, n'$);
4: REMOVE(n);

شکل ۳.۶: رویه WORKHORSE

فراخوانی مورد استفاده قرار می‌گیرد. این رویه را در شکل ۶.۶ شرح داده‌ایم. همان‌طور که مشاهده می‌شود، این رویه چندگانگی و شناسه هر گره را به‌صورت مناسبی تعیین می‌کند. توجه کنید که در تشریح این رویه و رویه CLEANRETNODES، از عملگر سطح بالای جایگذاری برای سادگی در نمایش و اجتناب از جزئیات پیاده‌سازی، استفاده کرده‌ایم.

رویه UNIFY آخرین رویه‌ای است که شرح می‌دهیم. این رویه که برای اتصال گره‌های گراف اشاره‌گری حاصل از خلاصه متد فراخوانی شده مورد استفاده قرار می‌گیرد، در شکل ۷.۶ شرح داده شده است. رویه MAKERETNODE(n) در آن، برای تبدیل گره n به یک گره برگشتی مورد استفاده قرار گرفته است. از آنجایی که مراحل این رویه بسیار شبیه به رویه تشریح شده در شکل ۵.۶ است، نیازی به فهرست کردن دقیق دستورالعمل‌ها را احساس نمی‌کنیم.

REMOVE$(n) \triangleq$

1: $ALoc' := ALoc' - \{n\};$

2: $ARef'_I := ARef'_I - \{(n', fd, n) \mid (n', fd, n) \in ARef'_I\};$

3: $ARef'_I := ARef'_I - \{(n, fd, n') \mid (n, fd, n') \in ARef'_I\};$

4: $ARef'_E := ARef'_E - \{(n', fd, n) \mid (n', fd, n) \in ARef'_E\};$

5: $ARef'_E := ARef'_E - \{(n, fd, n') \mid (n, fd, n') \in ARef'_E\};$

شکل ۴.۶: رویه REMOVE

CLEANRETNODES() $\triangleq$

1: $H' := H'\,[n/\mathtt{r}(n)]\,;$

2: $C' := C'\,[n/\mathtt{r}(n)]\,;$

شکل ۵.۶: رویه CLEANRETNODES

INSTANTIATE$(\ell) \triangleq$

1: $m' := getLabMult(\ell);$

2: $H' := H'\,[\mathtt{i}(\ell s, \ldots)/\mathtt{i}(s, \ldots)]\,[\mathtt{i}(\ldots, m \otimes m')/\mathtt{i}(\ldots, m)]\,;$

شکل ۶.۶: رویه INSTANTIATE

با وجود اینکه الگوریتم MAP در حالت کلی نادرست است، همان طور که مشاهده کردیم این الگوریتم تا حد ممکن جواب درستی را تخمین می‌زند. برای مثال، در خط اول این الگوریتم هنگامی که خلاصه متد فراخوانی شده در دسترس نباشد، با تولید نگاشت‌هایی در مخزن مجرد برای اتصال ℓ به گره‌هایی که هر یک از آرگومان‌ها ممکن است به آن‌ها اشاره کند، پاسخی محافظه‌کارانه تولید می‌کند.

تابع انتقال برای $[e.\mathtt{start}()]^{\ell}$: در تحلیل اشاره‌گر، رسیدگی به این حالت مانند حالت فراخوانی متد است، با این تفاوت که تابع $callees(\ell)$ این بار مجموعه‌ای از اسامی متدها به‌صورت $c.\mathtt{run}$ را برمی‌گرداند. الگوریتم نگاشت برای این دسته از عبارات، همانند فراخوانی متد است (البته با فرض $e \triangleq e.$).

۳.۲.۶ خلاصه متدها

بلوک‌های $begin$ و end، بلوک‌های ورودی و خروجی بدنه متدها هنوز شرح داده نشده است. تابع انتقال برای این بلوک‌ها تابع همانی است. بنابراین، توابع gen و $kill$ برای این بلوک‌ها برابر با $\bot_P$ خواهد بود.

همان‌طور که پیش‌تر نیز اشاره کردیم، خلاصه هر متد چیزی نیست جز یک گراف اشاره‌گری، چنین گرافی اثر متد مورد نظر بر روی گراف اشاره‌گری (در اصل اثر آن بر روی هیپ مجرد) متد فراخوانی کننده را بیان می‌کند. بنابراین، مؤلفه دوم یک مقدار اطلاعات اشاره‌گری می‌تواند به عنوان خلاصه متد مورد استفاده قرار گیرد. لازم به یادآوری است که تنها اطلاعات اشاره‌گری مربوط به گره خروجی یک متد در بر گیرنده تمامی اطلاعات از اثر متد فراخوانی شده است. در صورتی

```
UNIFY(n, n') ≜
1:  for each (n'', fd, n') ∈ ARef'_I do
2:      ARef'_I := ARef'_I ∪ {(n'', fd, n)};
3:  for each (n', fd, n'') ∈ ARef'_I do
4:      ARef'_I := ARef'_I ∪ {(n, fd, n'')};
5:  for each (n', fd, n'') ∈ ARef'_E do
6:      if int(n) then
7:          let N'' = {m | (n, fd, m) ∈ ARef_I} in
8:              WORKHORSE(N'', n'');
9:      else if ext(n) then
10:         let N'' = {m | (n, fd, m) ∈ ARef_I ∨ (n, fd, m) ∈ ARef_E} in
11:             if N'' = ∅ then
12:                 ARef'_E := ARef'_E ∪ {(n, fd, n'')};
13:             else
14:                 WORKHORSE(N'', n'');
15: if n' ∈ N_R then
16:     MAKERETNODE(n);
```

شکل ۷.۶: رویه UNIFY

که $p = PointsTo_\bullet(\ell)$ به شرطی که $[end(p_1, \ldots, p_k)]^\ell \in B$، با فرض B به عنوان مجموعه تمام بلوک‌های یک متد مانند $c.md$، باشد. همچنین π^1 و π^2 به ترتیب برون‌فکنی اول و دوم یک چندتایی باشد، خلاصه متد $c.md$ را به‌صورت زیر تعریف می‌کنیم:

$$ms(c.md) = \{\pi^2(\pi^1(p))\}.$$

با تحلیل هر متد در گراف فراخوانی برنامه، خلاصه آن را به نگاشت ms اضافه می‌کنیم. به این ترتیب با استفاده از ms گسترش یافته در تحلیل متدها به اطلاعات اشاره‌گری دقیق‌تری دست پیدا می‌کنیم. برای هر متد $c'.md'$ که هنوز تحلیل نشده است، مقدار ms برابر با $\emptyset$ تعریف می‌شود. بنابراین، نگاشت ms به عنوان یک تابع کامل به‌صورت زیر قابل تعریف است:

$$ms : Methods \to \mathcal{P}(PG).$$

۴.۲.۶ سیستم نوع

در این بخش، سیستم نوعی به منظور وارسی صحت حاشیه‌نوشته‌های دگرنامی ارائه می‌دهیم. این سیستم نوع با گسترش دو قاعده استنتاج مربوط به بررسی خوش‌فرمی صورت متدها و نیز فراخوانی متدها به دست می‌آید. در ادامه، شیوه گسترش هر یک از قواعد را شرح می‌دهیم.

گسترش قاعده استنتاج مربوط به خوش‌فرم بودن اعلان متدها. با توجه به نحو مجرد گسترش یافته در شکل ۱.۶، صورت یک متد در حالت کلی به‌صورت زیر اعلان می‌شود:

$$asdec\ t\ md(t_1\ var_1\ asa_1, \ldots, t_k\ var_k\ asa_k)asa.\ \{e\}$$

همان‌طور که مشاهده می‌کنیم، در این بخش صرفاً قواعد استنتاج را گسترش می‌دهیم. بنابراین، نیازمندی خوش‌فرم بودن نوع‌ها را دیگر بیان نمی‌کنیم، چرا که پیش‌تر در قاعده استاندارد بیان شده است. با مراجعه به نحو مجرد، در می‌یابیم که متغیر نحوی $asdec$ می‌تواند برابر با رشته تهی باشد، بنابراین نیازمندی مشروط خود را به‌صورت زیر بیان می‌کنیم. توجه کنید که نمادهای Λ و «$\cdot$»، در شرط‌های زیر به ترتیب نشان دهنده رشته تهی و عملگر الحاق رشته است.

$$asdec = \Lambda \implies asa. \cdot asa_1 \cdot \ \ldots\ \cdot asa_k = \Lambda$$

$$asdec = \texttt{aset}\ asid^* \implies$$
$$(\text{ASIDOnce}(asid^*) \wedge \text{WFAssignments}(asid^*, asa., asa_1, \ldots, asa_k))$$

که در آن گزاره ASIDOnce برای بررسی عدم تکرار در اعلان شناسه‌های مجموعه‌های دگرنامی به‌کار رفته است. همچنین، گزاره WFAssignments برای بررسی این موضوع که هر شناسه اعلان شده حتما در یک انتساب به‌کار رفته، و نیز برای بررسی این موضوع که در انتساب‌ها از شناسه‌های اعلان شده استفاده شده است. این دو گزاره را با استفاده از دو رویه در شکل‌های ۸.۶ و ۹.۶ توصیف کرده‌ایم. با هر بار فراخوانی رویه $tokenize$ در رویه ASIDOnce بزرگترین زیررشته از پیشوند s که حاوی ","‌ نیست را جدا کرده و به عنوان خروجی برمی‌گرداند. آرگومان s نیز به گونه‌ای بروزرسانی می‌شود که حاوی زیررشته مورد نظر نباشد.

ASIDOnce(s) $\triangleq$

```
1: S := ∅;
2: do
3:     if a ∈ S then return false;
       else S := S ∪ {a};
       a = tokenize(s, ",");
4: while a ≠ Λ;
5: return true;
```

شکل ۸.۶: رویه ASIDOnce

```
WFAssignments(s, q., q_1, ..., q_k) ≜
1: let a_1 · "," · ... · "," · a_n = s in
2:     S := {a_1, ..., a_n} × {•};
3: for each q ∈ {q., ..., q_k} do
4:     let (q = in a) in
5:         if a ∈ dom(S) then S(a) + +;
6:         else return false;
7: for each a ∈ dom(S) do
8:     if S(a) < 1 then return false;
9: return true;
```

شکل ۹.۶: رویه WFAssignments

گسترش قاعده استنتاج مربوط به فراخوانی متدها. با توجه به نحو مجرد ارائه شده در شکل ۱.۶، صورت کلی یک عبارت فراخوانی متد را می‌توان به‌صورت زیر نوشت:

$$e..md(e_1, \ldots, e_k)$$

قاعده استنتاج گسترش یافته ابتدا با بررسی نوع $e.$، (خانواده‌ای از) کلاسی (هایی) که قرار است متد md از آن‌ها بر روی $e.$ اعمال شود، مشخص می‌کند. خوشبختانه با توجه به این که صورت تمامی متدهای هم‌نام در یک خانواده از کلاس‌ها یکسان است، از دیدگاه قاعده استنتاج یاد شده تفاوتی بین نسخه‌های مختلف md وجود ندارد. بنابراین، فرض می‌کنیم که متد md به‌صورت زیر در کلاسی که معرف نوع t' است تعریف شده است.

$$e. : t'$$

$$asdec\ t\ md(t_1\ var_1\ asa_1, \ldots, t_k\ var_k\ asa_k)asa.\ \{e\} \in t'$$

که در آن $asdec$ اعلان شناسه‌های مجموعه‌های دگرنامی است که در این بخش با آن کاری نداریم، و به ازای هر $0 \leq i \leq k$ داریم: $asa_i = (\texttt{in}\ a_i)^?$.

حال تابع $asid$ را به‌صورت زیر برای استخراج شناسه مجموعه‌های دگرنامی از asa_iها تعریف می‌کنیم:

$$asid(s) = \begin{cases} a & ; s = \texttt{in}\ a \\ b & ; s = \Lambda \end{cases}$$

که در آن b یک شناسه مجموعه دگرنامی تازه است.

با توجه به تابع $asid$، مجموعه زیر را برای ارتباط هر یک از شناسه‌های مجموعه‌های دگرنامی پارامترها به مکان‌هایی که

آرگومان متناظر با آن پارامتر می‌تواند به آن اشاره کند، تعریف می‌کنیم:

$$A = \{(asid(asa_i), mayPointsTo(e_i)) \mid ۰ \leq i \leq k\}.$$

تابع کمکی زیر را نیز تعریف می‌کنیم:

$$f(A, a) = \bigcup\{P \mid (a, P) \in A\}$$

حال نیازمندی اصلی قاعده توسعه بافته به‌صورت زیر قابل بیان است:

$$\forall a_۱, a_۲ \in dom(A) \cdot \ a_۱ \neq a_۲ \implies f(A, a_۱) \cap f(A, a_۲) = \emptyset$$

این نیازمندی بیان می‌کند که دو مجموعه از مکان‌هایی که دو آرگومان متناظر با دو مجموعه دگرنامی متمایز به آن‌ها اشاره می‌کند، می‌بایست مجزا از هم باشد.

۵.۲.۶ نتیجه‌گیری

هدف اصلی از ارائه این بخش از پایان‌نامه، تعریف دقیق تابع $mayPointsTo$، برای تعیین مکان‌هایی که یک عبارت مانند e می‌تواند به آن اشاره کند، است. برای تعریف این تابع فرض می‌کنیم عبارت e به‌طور تحلیل اشاره‌گری شده و جواب $(PointsTo_\circ, PointsTo_\bullet, \mu)$ را برای آن در دست داریم. با این حساب تابع $cacheOf$ را برای استخراج مخزن مجرد به‌صورت زیر تعریف می‌کنیم:

$$cacheOf : PI^\star \to Cache$$

$$cacheOf(p) = \pi^۱(\pi^۲(p)).$$

تابع $mayPointsTo$ از مجموعه تمام عبارات برچسب دار به مجموعه‌ی گره‌ها را به‌صورت زیر تعریف می‌کنیم:

$$mayPointsTo(e) = \bigcup\{cacheOf(PointsTo_\bullet(\ell))(\ell) \mid \ell \in final(e)\}.$$

فصل هفتم

مقایسه با کارهای مرتبط

با مرور کمبودهای موجود در هر یک از کارهای معرفی شده، در فصل‌های اول و سوم، مشاهده کردیم که مقیاس‌پذیری یک ابزار کشف ایستای رقابت داده، در عین حفظ درستی و دقت، و بدون بالا رفتن میزان حاشیه‌نویسی یک نیاز در حوزه طراحی و ساخت این‌گونه ابزارهای صحت‌سنجی است. در این بخش، تلاش صورت گرفته در این پایان‌نامه با کارهای مشابه مقایسه می‌شود. به این ترتیب، میزان سهم پژوهش انجام شده در این پایان‌نامه در حل مشکلات موجود در زمینه طراحی و ساخت ابزارهای کشف ایستای رقابت داده روشن‌تر می‌شود.

همان‌طور که پیش‌تر نیز اشاره شد، موضوع این پایان‌نامه ارتباطی با روش‌های تست پویا و وارسی مدل ندارد. بنابراین، در این بخش از مقایسه کار خود با پژوهش‌های انجام شده در این حوزه خودداری کرده، و به مقایسه با روش‌های ایستا بسنده می‌کنیم. از طرفی روش ارائه شده در این پایان‌نامه، برخلاف روش‌های کاربردی، درست است. همچنین، بر خلاف روش‌های مبتنی بر نوع، سربار حاشیه‌نویسی بالایی ندارد و با این حال مقیاس‌پذیر است. در نتیجه، تنها به مقایسه کار خود با پژوهش‌های انجام شده در زمینه روش‌های مبتنی بر تحلیل جریان داده می‌پردازیم.

IBM: ابزار کشف رقابت داده مفهوم گراف جریان کنترل بین‌ریسه‌ای معادل مفهوم اجرای مجرد معرفی شده در این پایان‌نامه است. مجموعه رخدادها در یک اجرای مجرد معادل با گره‌های گراف جریان کنترل بین‌ریسه‌ای، و دوتایی‌های موجود در مؤلفه دوم یک اجرای مجرد معادل با جریان‌های کنترلی گراف یاد شده است. اما اجراهای مجرد به‌گونه‌ای طراحی شده‌اند که رخدادهای موجود آن‌ها اطلاعات بیشتری را به همراه خود دارند، که این در بهبود کیفیت گزارشات ارائه شده توسط ابزار نقشی اساسی دارد. تحلیل مورد استفاده برای استخراج اجراهای مجرد در این پایان‌نامه حساس به زمینه است، و بین فراخوانی‌های مختلف یک متد تفاوت قائل می‌شود. بنابراین، اطلاعات موجود در رخدادها حاوی اطلاعات مکان‌های فراخوانی نیز هست. این اطلاعات اضافی به برنامه‌نویس کمک می‌کند که آن دسته از فراخوانی‌هایی که موجب به وجود آمدن رقابت داده می‌شوند را از آن دسته از فراخوانی‌هایی که موجب بروز چنین خطایی نمی‌شوند تفکیک کند. بنابراین، برنامه‌نویس تنها در آن دسته از مکان‌های فراخوانی که امکان به وجود آمدن رقابت داده در آن‌ها وجود دارد از دستورات همگام‌سازی استفاده می‌کند. این‌گونه اطلاعات اضافی در گزارشات ارائه شده توسط یک ابزار کشف رقابت موجب می‌شود که بسیاری از دستورالعمل‌های همگام‌سازی تنها در مواقع نیاز به کار برده شوند، و با توجه به این‌که استفاده از دستورات

همگام‌سازی سربار محاسباتی دارد این کار موجب بهبود کارایی برنامه‌های تولید شده می‌شود.

مشکل اساسی تحلیل‌های ارائه شده در [۱۸]، مقیاس‌پذیر نبودن آن‌ها است. در این پایان‌نامه، تحلیلی پیمانه‌ای برای استخراج اجراهای مجرد از روی متن برنامه‌ها ارائه داده‌ایم، این تحلیل ضمن این‌که موجب مقیاس‌پذیر شدن کل سیستم کشف رقابت می‌شود، امکان تحلیل برنامه‌های باز را نیز به ما می‌دهد. برای بالا بردن دقت اطلاعات اشاره‌گری استخراج شده از برنامه‌های باز از حاشیه‌نوشته‌هایی در صورت متدها استفاده می‌کنیم.

سیستم کشف رقابت **ETH**: ابزار ارائه شده در [۲۳] شبیه‌ترین کار به پژوهش ارائه شده در این پایان‌نامه است. این سیستم، از مفهوم رخدادها و ترتیب زمانی بین آن‌ها برای کشف رقابت استفاده می‌کند. مفهوم اجراهای مجرد، که در این پایان‌نامه معرفی شده است، نیز بر اساس مفهوم رخدادها و ترتیب زمانی بین آن‌ها است. تفاوت بین اجراهای مجرد و گراف‌های استفاده اشیاء در این است که اجراهای مجرد مفهومی مستقل از زبان برنامه‌سازی بوده و امکان مدل‌سازی تمام روش‌های همگام‌سازی به جز روش‌های ابتکاری را به طراح تحلیل می‌دهد. این درحالی است که گراف استفاده اشیاء فقط برای زبان برنامه‌سازی جاوا طراحی شده و تنها قابلیت در نظر گرفتن همگام‌سازی به روش فقل‌گذاری و شروع و پایان ریسه‌ها را دارد.

این سیستم کشف رقابت درست نیست. نادرستی سیستم یاد شده بنا چند دلیل است که مهم‌ترین آن‌ها به این شرح است. رخدادهای موجود در گراف استفاده اشیاء منتسب به ریسه‌های مجرد هستند و هر ریسه مجرد با یک شی مجرد در گراف شکل هیپ مشخص می‌شود. از طرفی، از آنجایی که تحلیل اشاره‌گر مورد استفاده در سیستم یاد شده مبتنی بر کلاس هم‌ارزی است، امکان ادغام چندین شی ریسه و تشکیل یک شی مجرد (یک گره در گراف شکل هیپ) وجود دارد. بنابراین، امکان در نظر گرفتن چندین ریسه در زمان اجرا به عنوان یک ریسه مجرد وجود دارد. در این پایان‌نامه، به جای استفاده از گراف شکل هیپ برای تفکیک ریسه‌های مجرد مکان‌های ایجاد ریسه را به عنوان ریسه‌های مجرد در نظر می‌گیریم. همچنین، با استفاده از اطلاعات چندگانی مکان‌های ایجاد ریسه از امکان ادغام ریسه‌هایی که در حلقه‌های تکرار و توابع بازگشتی ایجاد می‌شوند جلوگیری به عمل می‌آید.

در نهایت با وجود این‌که سیستم ارائه شده در [۲۳] پیمانه‌ای و مقیاس‌پذیر است، تحلیل اشاره‌گری مورد استفاده در آن غیر دقیق، و تنها مناسب برای کشف رقابت شی، است. در این پایان‌نامه، به جای استفاده از یک تحلیل مبتنی بر کلاس هم‌ارزی، از یک تحلیل اشاره‌گری پیمانه‌ای حساس به زمینه دقیق‌تر، مشابه تحلیل‌های ارائه شده در [۳۶، ۴۷]، استفاده شده است.

ابزار **Relay**: ایده خلاصه متد مورد استفاده در این پایان‌نامه، برای دستیابی به مقیاس‌پذیری، مشابه خلاصه توابع در Relay است. با این تفاوت که در خلاصه متدها روابط زمانی بین رخدادها در نظر گرفته شده‌اند، در نتیجه استفاده از آن‌ها در تشخیص دقیق نقص‌های رقابت داده و اجتناب از گزارشات نادرست کمک می‌کند. همچنین، در این پایان‌نامه نشان داده‌ایم که چگونه می‌توان ضمن استفاده از خلاصه متدها درستی و دقت تحلیل را حفظ کرد.

همان‌طور که در فصل ۳ مشاهده کردیم، سمافورها یک منبع کاهش دقت و گزارشات نادرست در Relay است. در این پایان‌نامه مشاهده کردیم که با در نظر گرفتن یک رخداد به ازای هر عمل همگام‌سازی، می‌توان تحلیلی مستقل از اصطلاحات همگام‌سازی ساخت که در آن به راحتی می‌توان اثر سمافورها را تشخیص داد.

تحلیل اشاره‌گر مورد استفاده در Relay دقت کافی را در مدل‌سازی جریان‌های داده‌ای بین‌رویه‌ای نداشته و درست نیست [۳۵]. در این پایان‌نامه، ضمن ارائه یک تحلیل اشاره‌گر دقیق پیمانه‌ای نشان داده‌ایم که چگونه می‌توان حین کشف رقابت

داده ارتباط جریان‌های داده‌ای بین‌رویه‌ای مربوط به تحلیل اشاره‌گری را برقرار کرد.

اثباتی برای درستی Relay ارائه نشده است، و اساساً نادرست است. در این پایان‌نامه مشاهده کردیم که ارائه اثباتی مستقیم برای درستی تحلیل پیمانه‌ای کار دشواری است، بنابراین، اثبات درستی را از طریق ساخت یک تحلیل بین‌رویه‌ای، اثبات درستی آن، و در نهایت اثبات معادل بودن دو تحلیل پیمانه‌ای و بین‌رویه‌ای انجام داده‌ایم.

ابزار **Chord**: مفهوم رخدادها، که در این پایان‌نامه ارائه شده است، معادل مفهوم دستورالعمل‌ها در Chord است. دستورالعمل‌ها، همانند رخدادها، دارای اطلاعات زمینه فراخوانی هستند. بنابراین، خروجی ابزار Chord نیز حاوی اطلاعات سودمندی است.

دقت ابزار Chord بالا، سرعت آن قابل قبول، و نسخه نهایی آن نیز درست است. تنها مشکل جدی این ابزار (البته نسخه نهایی آن که درست است) مقیاس‌پذیر نبودن آن، به دلیل تحلیل کل برنامه به‌صورت یک‌جا، است [۲، ۴]. نتایج آزمایش‌ها، در [۲۴]، نشان می‌دهد که این ابزار قادر به تحلیل برنامه‌های بزرگتر از پنج هزار خط را ندارد. ما با پیاده‌سازی تحلیل پیمانه‌ای ارائه شده در این پایان‌نامه، با وجود اینکه هنوز از ساختمان داده‌های مناسب برای ذخیره‌سازی خلاصه متدها و نتایج سایر محاسبات استفاده نشده است، موفق به تحلیل برنامه‌هایی بزرگتر از ده هزار خط کد شده‌ایم.

ابزار ارائه شده در این پایان‌نامه، برخلاف Chord، برای تحلیل برنامه‌های باز نیازی به ساخت یک برنامه مشتری ندارد. در این روش، خلاصه متدها حاوی اطلاعات سودمندی در مورد الگوی دسترسی متدها است که می‌توان از آن برای کشف موارد وجود رقابت داده، در صورتی که متد مورد نظر حداقل یک ریسه ایجاد کند، استفاده کرد.

فصل هشتم

نتیجه‌گیری و کارهای آینده

هدف ما در این پایان‌نامه، ساخت یک سیستم پیمانه‌ای، درست، و با دقت بالا برای کشف رقابت داده است. منظور ما از پیمانه‌ای بودن این است که سیستم مورد نظر هر متد یک برنامه را جداگانه و بدون نیاز به کل برنامه تحلیل کند. شیوه معمول برای انجام تحلیل مبتنی بر جریان داده یک برنامه، عبارت است از ساخت گراف جریان کنترل بین‌رویه‌ای، تشکیل، و حل معادلات جریان داده برای کل برنامه. این درحالی است که در یک تحلیل پیمانه‌ای نیاز داریم گراف جریان کنترل و معادلات جریان داده برای هر متد برنامه را جداگانه تشکیل و حل کنیم. به عبارتی دیگر، در یک تحلیل پیمانه‌ای نیاز داریم که ارتباط بین متدهای برنامه را با مکانیزیمی به غیر از جریان‌های بین‌رویه‌ای برقرار کنیم. ما این کار را با استفاده از خلاصه متدها انجام می‌دهیم. جواب به‌دست آمده برای هر متد، که خلاصه متد نام دارد، پارامتری بوده و قابلیت این را دارد که در هر مکان فراخوانی متناسب با زمینه نمونه‌سازی شود. بر خلاف تحلیل‌های بین‌رویه‌ای معمول، در مکان‌های فراخوانی به جای تحلیل بدنه متدها خلاصه آن‌ها را نمونه‌سازی می‌کنیم. بنابراین، در این شیوه از تحلیل اطلاعات بین‌رویه‌ای به جای رد و بدل شدن از طریق جریان‌های بین‌رویه‌ای، با نمونه‌سازی خلاصه متدها در مکان‌های فراخوانی ساخته می‌شود. نمونه‌سازی یک خلاصه متد، با فرض n به عنوان تعداد دستورالعمل‌های موجود در آن متد، دارای پیچیدگی محاسباتی $O(n)$ است. حال آن‌که محاسبه مجدد جواب برای بدنه متد در هر مکان فراخوانی زمانی از مرتبه $n^۴$ لازم دارد.

ایده خلاصه متدها، در پژوهش‌های پیشین مانند [۲۱، ۲۳، ۱۴]، به‌عنوان راه‌حلی ساده و شهودی، که درستی تحلیل را تحت تأثیر قرار می‌دهد، برای افزایش مقیاس‌پذیری مورد استفاده قرار گرفته است. دستاورد ما ارائه تحلیلی درست مبتنی بر خلاصه متدها است. برای دستیابی به درستی، بر خلاف روش‌های پیمانه‌ای موجود، حین انجام تحلیل اجرای مجرد (برای محاسبه اطلاعات همروندی) ارتباط ارجاع‌ها در متدهای مختلف را برقرار کرده‌ایم. همچنین، به جای استفاده از گراف شکل هیپ برای شناسایی ریسه‌های مجرد، مکان‌های ایجاد ریسه در متن برنامه را به عنوان ریسه‌های مجرد در نظر گرفته‌ایم. در نهایت، با در نظر گرفتن چندگانگی برچسب‌ها از ادغام نمونه‌های مختلف ریسه‌های در زمان اجرا جلوگیری به عمل می‌آوریم. در اثبات درستی، مشاهده کردیم که استفاده از یک تحلیل بین‌رویه‌ای (غیر پیمانه‌ای) برای اثبات درستی و اثبات معادل بودن دو تحلیل بین‌رویه‌ای و پیمانه‌ای، بسیار ساده‌تر از اثبات مستقیم درستی برای تحلیل پیمانه‌ای است.

سیستم کشف رقابت داده پیشنهاد شده در این پایان‌نامه، مستقل از زبان برنامه‌سازی است و قابلیت تشخیص صحیح اصطلاحات همگام‌سازی مانند سمافورها را دارد. ایده ما برای فراهم کردن چنین قابلیتی، در نظر گرفتن رخدادهای مجزا

```
semaphore s = ۲;
resource r;

thread T = {
    wait(s);
    /*استفاده از منبع r*/
    signal(s);
}

parbegin T1, T2, T3, T4;
```

شکل ۱.۸: یک برنامه، در یک زبان فرضی، که در آن چهار ریسه به‌صورت همروند از یک منبع استفاده می‌کنند. در این برنامه، با استفاده از یک سمافور، ریسه‌ها طوری همگام‌سازی شده‌اند که در هر لحظه فقط دو ریسه توانایی دسترسی همروند به منبع مورد نظر را دارد.

برای هر یک از عملیات همگام‌سازی، به جای در نظر گرفتن مجموعه قفل‌های اخذ شده حین تولید هر رخداد است. برای مثال، برنامه شکل ۱.۸ را در نظر بگیرید که در آن r یک منبع است که در هر زمان حداکثر دو ریسه می‌توانند به‌صورت همروند از آن استفاده کنند. متغیر s نیز نشان دهنده یک سمافور با مقدار اولیه ۲ است. از ساختار parbegin برای ایجاد چهار ریسه، که به‌صورت همروند اجرا می‌شوند، استفاده شده است. در نهایت، بدنه ریسه‌ها در بلوک thread با شناسه T اعلان شده است. در این برنامه، برای چنین منبعی، خطای رقابت داده در زمان اجرا با حداقل سه رخداد متضاد (که متضاد بودن دو رخداد بسته به نوع منبع تعریف می‌شود)، به‌طوری که هیچ کدام از آن‌ها با هم رابطه پیش‌رویدادی ندارند، تعریف می‌شود.

در این برنامه، دنباله رخدادها برای هر یک از ریسه‌ها به‌صورت نشان داده شده در شکل ۲.۸ است. توجه کنید که در این شکل، رخداد از نوع P برای نشان دادن عمل wait، رخداد از نوع V برای نشان دادن عمل signal به‌کار رفته، و دنباله رخدادهای مربوط به دستکاری منبع مورد نظر را با چندنقطه مشخص کرده‌ایم. با توجه به این‌که مقدار سمافور s برابر با ۲ است، برای تعریف یک ترتیب ایمن بین رخدادها از ریسه‌های مختلف کافی است، برای مثال، هر یک از دو رخداد $e_۲$ و $e_۴$ را نسبت به رخدادهای $e_۵$ و $e_۷$ مرتب تعریف کنیم. در این صورت، می‌توان به راحتی این موضوع که در این مجموعه از رخدادها حداکثر دو رخداد وجود دارد که نسبت به هم رابطه پیش‌رویدادی ندارند و درنتیجه برنامه یاد شده ایمن است، را بررسی کرد.

$$
\begin{aligned}
\mathtt{T_1}: &\quad e_۱ = P(\mathtt{T_1}, \mathtt{s}), \ldots, e_۲ = V(\mathtt{T_1}, \mathtt{s}),\\
\mathtt{T_2}: &\quad e_۳ = P(\mathtt{T_2}, \mathtt{s}), \ldots, e_۴ = V(\mathtt{T_2}, \mathtt{s}),\\
\mathtt{T_3}: &\quad e_۵ = P(\mathtt{T_3}, \mathtt{s}), \ldots, e_۶ = V(\mathtt{T_3}, \mathtt{s}),\\
\mathtt{T_4}: &\quad e_۷ = P(\mathtt{T_4}, \mathtt{s}), \ldots, e_۸ = V(\mathtt{T_4}, \mathtt{s}).
\end{aligned}
$$

شکل ۲.۸: رخدادهای تولید شده توسط هر ریسه در برنامه شکل ۱.۸

گسترش و بهبود ابزار پیاده‌سازی شده در این پایان‌نامه، به‌طوری که مناسب استفاده در صنعت باشد، را به عنوان پروژه‌ای برای آینده پیشنهاد می‌کنیم. برای این منظور، نیاز است که کارهای زیر انجام شود:

- پیاده‌سازی تحلیل اشاره‌گر پیمانه‌ای ارائه شده در این پایان‌نامه برای افزایش دقت اطلاعات اشاره‌گری،
- پیاده‌سازی یک تحلیل فرار پیمانه‌ای،
- بهبود کارایی ساختمان داده‌های مورد استفاده در پیاده‌سازی، و افزودن قابلیت استفاده از حافظه جانبی،
- گسترش نحو زبان جاوا، به شیوه پیشنهاد شده در این پایان‌نامه، برای افزایش دقت تحلیل‌ها اشاره‌گر و افزودن قابلیت تحلیل برنامه‌های باز به سیستم.

تحلیل فرار ارائه شده در [۳۶]، تحلیلی است که علاوه بر پیمانه‌ای بودن، قابلیت این را دارد که همزمان با تحلیل اشاره‌گر انجام شود. برای گسترش نحو زبان و نیز گسترش قواعد وارسی نوع متناسب با نحو گسترش یافته می‌توان، به جای Soot، از چهارچوب کامپایلری Polyglot استفاده کرد.

کتاب‌نامه

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley Professional, 2005. 4th Edition.

[2] M. Naik, *Effective Static Race Detection for Java*. Ph.D. thesis, Stanford University, March 2008.

[3] C. von Praun, "Race conditions," in *Encyclopedia of Parallel Computing* (D. A. Padua, ed.), pp.1691–1697, Springer US, 2011.

[4] C. von Praun, "Race detection techniques," in *Encyclopedia of Parallel Computing* (D. A. Padua, ed.), pp.1697–1706, Springer US, 2011.

[5] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, (New York, NY, USA), pp.387–400, ACM, 2012.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol.21, pp.558–565, July 1978.

[7] C. von Praun, *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. Ph.D. thesis, ETH Zurich, 2004.

[8] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*, vol.15, pp.391–411, Nov. 1997.

[9] C. von Praun and T. R. Gross, "Object race detection," in *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, (New York, NY, USA), pp.70–82, ACM, 2001.

[10] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, (New York, NY, USA), pp.167–178, ACM, 2003.

[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Communications of the ACM*, vol.53, pp.66–75, Feb. 2010.

[12] N. E. Beckman, "A survey of methods for preventing race conditions," tech. rep., Carnegie Mellon University, 2006.

[13] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol.17, pp.549–557, Oct. 1974.

[14] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp.237–252, ACM, 2003.

[15] N. Sterling, "WARLOCK - A static data race analysis tool," in *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*, pp.97–106, 1993.

[16] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Software*, vol.25, no.5, pp.22–29, 2008.

[17] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[18] J.-d. Choi, A. Loginov, and V. Sarkar, "Static datarace analysis for multithreaded object-oriented programs," tech. rep., IBM Research Division, Thomas J. Watson Research Centre, 2001.

[19] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), pp.308–319, ACM, 2006.

[20] M. Naik and A. Aiken, "Conditional must not aliasing for static race detection," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, (New York, NY, USA), pp.327–338, ACM, 2007.

[21] J. W. Voung, R. Jhala, and S. Lerner, “Relay: Static race detection on millions of lines of code,” in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE ’07, (New York, NY, USA), pp.205–214, ACM, 2007.

[22] E. Duesterwald and M. L. Soffa, “Concurrency analysis in the presence of procedures using a data-flow framework,” in *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV4, (New York, NY, USA), pp.36–48, ACM, 1991.

[23] C. von Praun and T. R. Gross, “Static conflict analysis for multi-threaded object-oriented programs,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI ’03, (New York, NY, USA), pp.115–128, ACM, 2003.

[24] A. Milanova and W. Huang, “Static object race detection,” in *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*, pp.255–271, 2011.

[25] L. Cardelli, “Type systems,” in *The Computer Science and Engineering Handbook* (A. B. Tucker, ed.), chap. 97, CRC Press, 2004.

[26] L. Cardelli and P. Wegner, “On understanding types, data abstraction, and polymorphism,” *ACM Computing Surveys*, vol.17, pp.471–523, Dec. 1985.

[27] B. C. Pierce. *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.

[28] C. Flanagan and M. Abadi, “Types for safe locking,” in *Programming Languages and Systems* (S. Swierstra, ed.), vol.1576 of *Lecture Notes in Computer Science*, pp.91–108, Springer Berlin Heidelberg, 1999.

[29] C. Flanagan and M. Abadi, “Object types against races,” in *CONCUR’99 Concurrency Theory* (J. Baeten and S. Mauw, eds.), vol.1664 of *Lecture Notes in Computer Science*, pp.288–303, Springer Berlin Heidelberg, 1999.

[30] C. Flanagan and S. N. Freund, “Type-based race detection for java,” in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI ’00, (New York, NY, USA), pp.219–232, ACM, 2000.

[31] C. Flanagan and S. N. Freund, “Detecting race conditions in large programs,” in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE ’01, (New York, NY, USA), pp.90–96, ACM, 2001.

[32] M. Abadi, C. Flanagan, and S. N. Freund, “Types for safe locking: Static race detection for java,” *ACM Transactions on Programming Languages and Systems*, vol.28, pp.207–255, Mar. 2006.

[33] C. Boyapati and M. Rinard, “A parameterized type system for race-free java programs,” in *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA ’01, (New York, NY, USA), pp.56–69, ACM, 2001.

[34] C. Boyapati, R. Lee, and M. Rinard, “Ownership types for safe programming: Preventing data races and deadlocks,” in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA ’02, (New York, NY, USA), pp.211–230, ACM, 2002.

[35] P. Pratikakis, J. S. Foster, and M. Hicks, “LOCKSMITH: practical static race detection for C,” *ACM Transactions on Programming Languages and Systems*, vol.33, no.1, p.3, 2011.

[36] J. Whaley and M. Rinard, “Compositional pointer and escape analysis for java programs,” in *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA ’99, (New York, NY, USA), pp.187–206, ACM, 1999.

[37] M. Vaziri, F. Tip, and J. Dolby, “Associating synchronization constraints with data in an object-oriented language,” in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’06, (New York, NY, USA), pp.334–345, ACM, 2006.

[38] R. Grimaldi. *Discrete and combinatorial mathematics - an applied introduction*. Addison-Wesley, 1993. 3rd Edition.

[39] D. Bjørner. *Software Engineering, vol. 1: Abstraction and Modelling*, vol.1 of *Texts in Theoretical Computer Science*. Springer, 2006.

[40] Wikipedia, “Zorn’s lemma,” 2014. [accessed 02-Sep-2014].

[41] P. Linz. *An Introduction to Formal Language and Automata*. USA: Jones and Bartlett Publishers, Inc., 2006.

[42] M. Flatt, S. Krishnamurthi, and M. Felleisen, “Classes and mixins,” in *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, POPL ’98, (New York, NY, USA), pp.171–183, ACM, 1998.

[43] R. H. B. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Letters on Programming Languages and Systems*, vol.1, pp.74–88, Mar. 1992.

[44] Sable, "Soot: a java optimization framework," `http://www.sable.mcgill.ca/soot/`, [accessed 18-Sep-2014].

[45] Á. Einarsson and J. D. Nielsen, "A survivor's guide to java program analysis with soot," tech. rep., Basic Research in Computer Science, Aarhus University. `http://www.brics.dk/SootGuide/`, [accessed 18-Sep-2014].

[46] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallèe-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," in *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, (New York, NY, USA), pp.264–280, ACM, 2000.

[47] M. Sagiv, T. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updates," *ACM Transactions on Programming Languages and Systems*, vol.20, pp.1–50, January 1998.

پیوست آ

معناشناخت ایستا

در این بخش، معناشناخت ایستا زبان جاوای همروند را با استفاده از یک سیستم نوع بیان می‌کنیم. این سیستم نوع را سیستم نوع استاندارد می‌گوییم. در هر یک از بخش‌های مختلف پایان‌نامه مانند تحلیل شکل هیپ و نیز تحلیل رخدادها، سیستم نوع استاندارد مطابق با نیاز گسترش داده می‌شود. برای تعریف این سیستم، ابتدا لازم است تعدادی گزاره معرفی کنیم. این گزاره‌ها به همراه توضیحات در جدول ۱.۱ فهرست شده است. برای اطلاع از تعریف دقیق گزاره‌های یاد شده، خواننده می‌تواند به مقاله [۴۲] مراجعه نماید.

جدول ۱.۱: گزاره‌های لازم برای تعریف سیستم نوع.

گزاره	توضیحات
$ClassOnce(P)$	در برنامه P هیچ کلاسی دو بار تعریف نشده است.
$WFClasses(P)$	هیچ سیکلی در سلسله مراتب کلاس‌های تعریف شده در P وجود ندارد.
$FieldsOnce(P)$	هیچ کلاسی در برنامه P، حاوی دو فیلد (اعلان شده، و یا به ارث برده شده) با یک نام نیست.
$MethodsOncePerClass(P)$	هیچ کلاسی در برنامه P، دو متد با یک نام اعلان نکرده است.
$OverridesOK(P)$	متدهای سربارگذاری شده دارای نوع برگشتی، تعداد و نوع پارامتر یکسان هستند.

محیط انتساب نوع E را به‌صورت زیر تعریف می‌کنیم:

$$E ::= \emptyset \mid E, var : t$$

که در آن var شناسه یک متغیر شامل شناسه ویژه `this` است. برای یک محیط انتساب نوع، تعریف‌های کمکی زیر را نیز ارائه می‌دهیم:

$$var : t \in E \iff E = E_1, var : t, E_2$$

$$dom(E) = \{var \mid \exists t \cdot var : t \in E\}$$

سیستم نوع تعریف شده در این بخش، حکم‌هایی به فرم ارائه شده در جدول زیر را اثبات می‌کند.

جدول ۲.۱: حکم‌ها

حکم	توضیحات
$\vdash P : t$	برنامه P دارای نوع t است.
$P \vdash defn$	اعلان کلاس $defn$ خوش فرم است.
$P; E \vdash \diamond$	محیط انتساب نوع E خوش فرم است.
$P; E \vdash defn$	کلاس $defn$ در برنامه P تعریف شده است.
$P; E \vdash t$	نوع t خوش فرم است.
$P; E \vdash t_1 <: t_2$	نوع t_1 زیرنوع t_2 است.
$P; E \vdash field \in t$	فیلد $field$ در کلاس t تعریف (اعلان شده یا به ارث برده شده) شده است.
$P; E \vdash field$	اعلان فیلد $field$ خوش فرم است.
$P; E \vdash meth \in t$	متد $meth$ در کلاس t تعریف (اعلان شده یا به ارث برده شده) شده است.
$P; E \vdash meth$	اعلان متد $meth$ خوش فرم است.
$P; E \vdash e : t$	در برنامه P و محیط انتساب نوع E، عبارت e دارای نوع t است.

اصول و قواعد استنتاج ارائه شده در ادامه این بخش، حکم‌های فهرست شده در جدول ۲.۱ را به‌صورت استقرایی تعریف می‌کند.

$\boxed{\vdash P : t}$

$$\text{(Prog)} \quad \frac{\begin{array}{c} ClassOnce(P) \qquad WFClasses(P) \\ FieldsOnce(P) \qquad MethodsOncePerClass(P) \\ OverridesOK(P) \\ P = defn_{1\ldots n}\ e \qquad P \vdash defn_i \qquad P; \emptyset \vdash e : t \\ i \in 1 \ldots n \end{array}}{\vdash P : t}$$

$\boxed{P \vdash defn}$

$$\text{(WF Class)} \quad \frac{\begin{array}{c} E = \texttt{this} : c \\ P; E \vdash field_i \qquad P; E \vdash meth_j \\ P; E \vdash c' \qquad i \in 1 \ldots n \qquad j \in 1..m \end{array}}{P \vdash \texttt{class}\ c\ \texttt{extends}\ c' \{field_{1\ldots n}\ meth_{1\ldots m}\}}$$

$\boxed{P; E \vdash \diamond}$

$$\text{(Env Empty)} \quad \frac{}{P; \emptyset \vdash \diamond} \qquad \text{(Env Var)} \quad \frac{P; E \vdash t \qquad var \notin dom(E)}{P; E, var : t \vdash \diamond}$$

$\boxed{P; E \vdash defn}$

$$\text{(Class Defined)} \quad \frac{P; E \vdash \diamond \qquad \texttt{class}\ c\ \ldots\ \in P}{P; E \vdash \texttt{class}\ c\ \ldots}$$

$\boxed{P; E \vdash t}$

$$\text{(Type Class)} \quad \frac{P; E \vdash \texttt{class}\ c\ \ldots}{P; E \vdash c} \qquad \text{(Type Object)} \quad \frac{P; E \vdash \diamond}{P; E \vdash \texttt{Object}}$$

$\boxed{P; E \vdash t_1 <: t_2}$

(Sub Refl) $\dfrac{P; E \vdash t}{P; E \vdash t <: t}$ (Sub/Tran Class) $\dfrac{P; E \vdash c_1 <: c_2 \quad P; E \vdash \texttt{class}\ c_2\ \texttt{extends}\ c_3\ \ldots}{P; E \vdash c_1 <: c_3}$

$\boxed{P; E \vdash field \in t}$

(Field Decl) $\dfrac{P; E \vdash \texttt{class}\ c\ \ldots\ \{\ldots\ field\ \ldots\}}{P; E \vdash field \in c}$ (Field Inh) $\dfrac{P; E \vdash t_1 <: t_2 \quad P; E \vdash field \in t_2}{P; E \vdash field \in t_1}$

$\boxed{P; E \vdash field}$

(WF Field) $\dfrac{P; E \vdash t}{P; E \vdash t\ fd}$

$\boxed{P; E \vdash meth \in t}$

(Meth Decl) $\dfrac{P; E \vdash \texttt{class}\ c\ \ldots\ \{\ldots\ meth\ \ldots\}}{P; E \vdash meth \in c}$ (Meth Inh) $\dfrac{P; E \vdash t_1 <: t_2 \quad P; E \vdash meth \in t_2}{P; E \vdash meth \in t_1}$

$\boxed{P; E \vdash meth}$

(WF Method) $\dfrac{P; E \vdash t \quad P; E, var_1 : t_1, \ldots, var_k : t_k \vdash e : t' \quad P; E \vdash t' <: t}{P; E \vdash t\ md(t_1\ var_1, \ldots, t_k\ var_k)\{e\}}$

$\boxed{P; E \vdash e : t}$

(Exp Sub) $\dfrac{P; E \vdash e : t \qquad P; E \vdash t <: s}{P; E \vdash e : s}$ (Exp New) $\dfrac{P; E \vdash t}{P; E \vdash \texttt{new}\ t : t}$ (Exp Var) $\dfrac{P; E \vdash \diamond \quad var : t \in E}{P; E \vdash var : t}$

(Exp Field Rd) $\dfrac{P; E \vdash e : t' \quad P; E \vdash (t\ fd) \in t'}{P; E \vdash e.fd : t}$ (Exp Field Wr) $\dfrac{P; E \vdash e : t \qquad P; E \vdash e' : t' \quad P; E \vdash (t_f\ fd) \in t \quad P; E \vdash t' <: t_f}{P; E \vdash (e.fd = e') : t'}$

(Exp Meth Call) $\dfrac{P; E \vdash e. : t. \quad P; E \vdash (t\ md(t_{1..k}\ var_{1..k})\{e\}) \in t. \quad P; E \vdash e_i : t'_i \qquad P; E \vdash t'_i <: t_i \quad i \in 1..k}{P; E \vdash e..md(e_{1..k}) : t}$ (Val Null) $\dfrac{P; E \vdash t}{P; E \vdash \texttt{null} : t}$

(Exp Sync) $\dfrac{P; E \vdash e_1 : t_1 \qquad P; E \vdash e_2 : t_2}{P; E \vdash \texttt{synchronized}\ e_1\ \texttt{in}\ e_2 : t_2}$ (Exp Let) $\dfrac{P; E \vdash e_1 : t_1 \qquad P; E, var : t_1 \vdash e_2 : t_2}{P; E \vdash \texttt{let}\ var = e_1\ \texttt{in}\ e_2 : t_2}$

(Exp Fork) $\dfrac{P; E \vdash e : t' \qquad P; E \vdash (t\ \texttt{run}()\{e'\}) \in t'}{P; E \vdash e.\texttt{start}() : t}$ (Exp Cast) $\dfrac{P; E \vdash e : t' \quad (P; E \vdash t <: t' \vee P; E \vdash t' <: t)}{P; E \vdash (t)e : t}$

پیوست ب

درستی تحلیل بین‌رویه‌ای

همان‌طور که در فصل ۴ مشاهده شد، پاسخ محاسبه شده توسط تحلیل‌های پیمانه‌ای و بین‌رویه‌ای، از دیدگاه کشف رقابت داده، با هم معادل هستند. بنابراین، اثبات درستی یکی درستی دیگری را نیز در پی دارد. در این بخش، درستی تحلیل بین‌رویه‌ای را که ساده‌تر است ثابت می‌کنیم. برای این منظور، نیاز داریم ابتدا معناشناخت عملیاتی زبان جاوای همروند را، که در ابتدای فصل ۴ معرفی کردیم، ارائه دهیم. برای این‌که بتوانیم یک معناشناخت عملیاتی مبتنی بر جایگذاری ارائه دهیم، نیاز داریم مطابق [۳۲] نحو زبان را با استفاده از آدرس‌ها و ساختار $insync$ گسترش دهیم:

$$e ::= \cdots \mid p \mid insync\ p\ in\ e$$

که در آن p نشان دهنده آدرس، یا همان یک شناسه برای یک شی (که در ادامه به‌صورت صوری تشریح می‌شود)، و ساختار $insync\ p\ in\ e$ به این معنا است که قفل مربوط به شی p اخذ شده و عبارت e در حال ارزیابی است. توجه کنید که این دو ساختار جدید نباید در متن برنامه‌ها وجود داشته باشند، این ساختارها در عبارت‌های میانی، و در اثر ارزیابی، به وجود می‌آیند.

شکل ۱.۲، معناشناخت صوری زبان جاوای همروند را با استفاده از یک ماشین مجرد توصیف می‌کند. ماشین یاد شده، یک برنامه را با گذر از دنباله‌ای از حالات ارزیابی می‌کند. یک حالت با استفاده از یک زوج مرتب توصیف شده است، که اولین مؤلفه آن مخزن اشیاء (عضوی از مجموعه $Store$)، و مؤلفه دوم آن دنباله‌ای از زوج‌های مرتب است؛ مؤلفه اول این زوج‌های مرتب شناسه ریسه، و مؤلفه دوم آن‌ها عبارتی (از نحو گسترش یافته) است که ریسه مورد نظر در حال ارزیابی آن است. از آن‌جایی که در زمان اجرا چندگانی تمامی ریسه‌ها (و شی‌ها) برابر یک است، از نوشتن چندگانگی در کنار شناسه ریسه‌ها (و شی‌ها) خودداری می‌کنیم. نتیجه ارزیابی یک برنامه برابر با نتیجه ارزیابی عبارت مربوط به ریسه اصلی (`main`) است، که همواره به عنوان مؤلفه دوم از اولین عنصر در دنباله ریسه‌های ظاهر می‌شود. ریسه‌های جدیداً ایجاد شده به انتهای دنباله ریسه‌ها افزوده می‌شوند. در صورتی که T_1 و T_2 دو دنباله از ریسه‌ها (زوج شناسه ریسه و عبارت مربوط به ریسه یاد شده) باشند، از $T_1.T_2$ برای نشان دادن الحاق این دو دنباله استفاده می‌کنیم.

در هر حالت، شی‌ها در یک مخزن اشیاء $\sigma \in Store$ نگهداری می‌شود. یک مخزن اشیاء نگاشتی از آدرس‌ها به شی‌ها

ارزیابی برنامه:

$$eval(P,v) \iff P \vdash \langle \emptyset, (\texttt{main}, e)\rangle \hookrightarrow^* \langle \sigma, (\texttt{main}, v_1), \ldots, (\tau_n, v_n)\rangle\,, \quad P = defn^*\ e \text{ که در آن}$$

فضای حالات:

$$\begin{aligned}
&S \in State = Store \times ThreadSeq,\\
&\sigma \in Store = \{s \mid s : Addresses \to Objects\},\\
&T \in ThreadSeq = (\tau, e)^*,\\
&\tau \in TID,\\
&o \in Objects = \{(\!|db|\!)_c^m \mid db \in (fn_i = v_i)^* \wedge m \in \{\mathsf{locked}, \mathsf{unlocked}\}\},\\
&p \in Addresses = \{p., p_1, \ldots\}.
\end{aligned}$$

زمینه‌های ارزیابی:

$$\begin{aligned}
\mathcal{E} ::= {} & \{\} \mid \mathcal{E}.fn \mid \mathcal{E}.fn = e \mid p.fn = \mathcal{E}\\
& \mid \mathcal{E}.mn(e^*) \mid p.mn(v^*, \mathcal{E}, e^*) \mid \texttt{let}\ var = \mathcal{E}\ \texttt{in}\ e\\
& \mid \mathcal{E}.\texttt{start}() \mid \texttt{synchronized}\ \mathcal{E}\ \texttt{in}\ e \mid insync\ p\ in\ \mathcal{E} \mid (t)\mathcal{E}
\end{aligned}$$

قواعد انتقال حالت:

[Red New]

$$P \vdash \langle \sigma, T.(\tau, \mathcal{E}\,[\texttt{new}\ c()]).T'\rangle \hookrightarrow \left\langle \sigma\left[(\!|db|\!)_c^{\mathsf{unlocked}}/p\right], T.(\tau, \mathcal{E}\,[p]).T'\right\rangle$$
$$P; c \vdash_{init} db \text{ و } p \notin dom(\sigma) \text{ به‌طوری که}$$

[Red Read]

$$P \vdash \langle \sigma, T.(\tau, \mathcal{E}\,[p.fn]).T'\rangle \hookrightarrow \langle \sigma, T.(\tau, \mathcal{E}\,[v]).T'\rangle$$
$$\sigma(p) = (\!|\ldots, fn = v, \ldots|\!)_c^m \text{ به‌طوری که}$$

[Red Assign]

$$P \vdash \langle \sigma, T.(\tau, \mathcal{E}\,[p.fn = v]).T'\rangle \hookrightarrow \langle \sigma\,[v/p.fn], T.(\tau, \mathcal{E}\,[v]).T'\rangle$$

[Red Call]

$$P \vdash \langle \sigma, T.(\tau, \mathcal{E}\,[p.mn(v_{1\ldots n})]).T'\rangle \hookrightarrow \left\langle \sigma, T.(\tau, \mathcal{E}\left[e\left[v_i/x_i{}^{i\in\{1\ldots n\}}, p/\texttt{this}\right]\right]).T'\right\rangle$$
$$P; \emptyset \vdash t\ mn(t_i\ x_i{}^{i\in\{1\ldots n\}})\{e\} \in c \text{ و } \sigma(p) = (\!|db|\!)_c^m \text{ به‌طوری که}$$

[Red Let]

$$P \vdash \langle \sigma, T.(\tau, \mathcal{E}\,[\texttt{let}\ var = v\ \texttt{in}\ e]).T'\rangle \hookrightarrow \langle \sigma, T.(\tau, \mathcal{E}\,[e\,[v/var]]).T'\rangle$$

[Red Sync]

$$P \vdash \langle \sigma, T.(\tau, \mathcal{E}\,[\texttt{synchronized}\ p\ \texttt{in}\ e]).T'\rangle \hookrightarrow \left\langle \sigma\left[(\!|db|\!)_c^{\mathsf{locked}}/p\right], T.(\tau, \mathcal{E}\,[insync\ p\ in\ e]).T'\right\rangle$$
$$\sigma(p) = (\!|db|\!)_c^{\mathsf{unlocked}} \text{ به‌طوری که}$$

[Red InSync]

$$P \vdash \langle \sigma, T.(\tau, \mathcal{E}\,[insync\ p\ in\ v]).T'\rangle \hookrightarrow \left\langle \sigma\left[(\!|db|\!)_c^{\mathsf{unlocked}}/p\right], T.(\tau, \mathcal{E}\,[v]).T'\right\rangle$$
$$\sigma(p) = (\!|db|\!)_c^{\mathsf{locked}} \text{ به‌طوری که}$$

[Red Start]

$$P \vdash \langle \sigma, T.(\tau, \mathcal{E}\,[p.\texttt{start}()]).T'\rangle \hookrightarrow \langle \sigma, T.(\tau, \mathcal{E}\,[\texttt{null}]).T'.(\tau', p.\texttt{run}())\rangle$$
$$\text{به‌طوری که } \tau' \neq \tau \text{ و } \tau' \text{ در } T.T' \text{ ظاهر نشود}$$

[Red Cast]

$$P \vdash \langle \sigma, T.(\tau, \mathcal{E}\,[(t)\ v]).T'\rangle \hookrightarrow \langle \sigma, T.(\tau, \mathcal{E}\,[v]).T'\rangle$$
$$v = \texttt{null} \text{ یا } P; \emptyset \vdash c <: t \text{ به‌طوری که } \sigma(p) = (\!|db|\!)_c^m \text{ که در آن}$$

شکل ۱.۲: معناشناخت صوری زبان ConcurrentJava$^+$

است. یک شی $(\!|db|\!)_c^m$ دارای سه مؤلفه است: مؤلفه db نگاشتی از شناسه فیلدها به مقادیر (آدرس‌ها یا null)، مؤلفه m نشان دهنده وضعیت قفل شی مورد نظر، و c کلاس شی را مشخص می‌کند. نگاشت شناسه فیلدها یا همان db یک فهرست به‌صورت $fn_1 = v_1, \ldots, fn_n = v_n$ است. از نماد Λ برای نشان دادن یک نگاشت خالی فیلدها (فهرست خالی یعنی شی مورد نظر هیچ فیلدی ندارد) استفاده می‌کنیم. وضعیت قفل m، برای یک شی، در یکی از دو حالت اخذ شده (locked) یا

آزاد (unlocked) قرار دارد.

ماشین مجرد مورد استفاده در این پایان‌نامه در هر گذار، به‌طور دلخواه و غیر قطعی، یک ریسه را برای ارزیابی عبارت مربوط به آن انتخاب می‌کند. واضح است که چنین انتخابی مانند یک زمان‌بندی دلخواه برای اجرای (در هم تنیده دستورالعمل‌های مربوط به) ریسه‌های یک برنامه است. بنابراین، مقدار حاصل از ارزیابی یک برنامه وابسته به ترتیب ارزیابی دستورالعمل‌های ریسه‌ها است. به همین دلیل، $eval$ به عنوان یک رابطه تعریف کردیم، چرا که یک برنامه P ممکن است به ازای زمان‌بندی‌های مختلف به مقادیر مختلفی ارزیابی شود.

اجرای یک برنامه با یک مخزن خالی اشیاء و تنها یک ریسه، و عبارت مربوط به آن که همان عبارت آغازین برنامه است، شروع می‌شود. ارزیابی، با توجه به قواعد انتقال حالت انجام می‌شود و اجرای برنامه زمانی خاتمه می‌یابد که عبارت مربوط به تمام ریسه‌های آن طی ارزیابی به مقادیر (یعنی آدرس‌ها و `null`) تبدیل شوند. از $\sigma\,[o/p]$، برای نشان دادن مخزنی که به ازای تمام آدرس‌ها به جز با σ برابر بوده، و مقدار آن در p برابر با o است. همچنین، از نماد $\sigma\,[v/p.fd]$ برای دادن مخزنی که به ازای تمام آدرس‌ها به جز با σ برابر بوده، و مقدار آن در p برابر با $\sigma(p)$ است، به‌قسمی که فیلد fd آن برابر با مقدار v است.

قاعده انتقال حالت [Red Start] برای $p.\texttt{start}()$ یک ریسه جدید برای ارزیابی $p.\texttt{run}()$ ایجاد کرده، و مقدار `null` را به عنوان نتیجه ارزیابی عبارت ایجاد ریسه تعیین می‌کند. قاعده [Red Sync] با ارزیابی `synchronized` p `in` e و تبدیل آن به عبارت $insync\ p\ in\ e$ قفل مربوط به شی p را اخذ می‌کند. عبارت جدید نشان دهنده این است که قفل شی p اخذ شده و عبارت e آماده ارزیابی است. بعد از این‌که عبارت e به‌طور کامل ارزیابی و تبدیل به یک مقدار مقدار مانند v شد، قاعده [Red InSync] ضمن آزاد سازی قفل منتسب به شی p، مقدار v را به عنوان نتیجه کل عبارت تعیین می‌کند. سایر قواعد انتقال حالت ساده بوده، و نیازی به توضیح بیش‌تر احساس نمی‌شود.

قواعد یاد شده بر اساس چند حکم کمکی تعریف شده‌اند. قاعده [Red New] از حکم کمکی $P; c \vdash_{init} db$ برای تعیین نگاشت اولیه فیلدها برای یک شی تازه ایجاد شده استفاده می‌کند. سایر قواعد از حکم‌های تعریف شده در سیستم نوع ارائه شده در پیوست اول پایان‌نامه استفاده می‌کنند. اصول و قواعد استنتاج ارائه شده در شکل ۲.۲ اثبات حکم $P; c \vdash_{init} db$ (یعنی db نگاشت اولیه فیلدهای شی‌های تازه ایجاد شده از روی کلاس c است) را به‌صورت بازگشتی توصیف می‌کنند.

$$(۱)\ \frac{}{P; \texttt{Object} \vdash_{init} \Lambda}, \qquad (۲)\ \frac{\begin{array}{c} P; \emptyset \vdash \texttt{class}\ c\ \texttt{extends}\ c'\{field_{1\ldots m}\ meth_{1\ldots n}\} \\ field_i = t_i\ fd_i \qquad i \in 1\ldots m \\ P; c' \vdash_{init} db \end{array}}{P; c \vdash_{init} db, fd_1 = \texttt{null}, \ldots, fd_m = \texttt{null}}$$

شکل ۲.۲: توصیف استقرایی اثبات حکم کمکی مورد استفاده در ماشین مجرد ارائه شده در شکل ۱.۲

گوییم عبارت e داخل **ناحیه بحرانی** (محافظت شده توسط) p است، هرگاه $e = \mathcal{E}\,[insync\ p\ in\ e']$، به ازای یک زمینه ارزیابی $\mathcal{E}$ و عبارت e'، باشد. از همین معناشناخت برای تعریف صوری مفهوم نقص رقابت داده استفاده می‌کنیم. گوییم عبارت e از ریسه τ به مکان حافظه $p.fn$ (فیلد fn از شی نشان داده شده توسط p) **دسترسی** دارد، هرگاه به ازای یک زمینه ارزیابی $\mathcal{E}$ و مقدار v داشته باشیم $e = \mathcal{E}\,[p.fn]$ یا $e = \mathcal{E}\,[p.fn = v]$ و e عبارت مربوط به ریسه τ در یک حالت باشد. در حالت اول گوییم دسترسی از نوع **خواندن** از مکان $p.fn$ است، و در حالت دوم گوییم دسترسی از نوع **نوشتن** در آن مکان است. یک حالت حاوی **دسترسی‌های متضاد** در مکان $p.fn$ است، هرگاه دنباله ریسه‌های آن حالت حاوی دو

یا چند عبارت (سطح بالا: عبارتی که در یک زمینه ارزیابی سوراخ $\{\}$ با آن پر می‌شود) است که به مکان $p.fn$ دسترسی دارند، و حداقل یکی از آن‌ها از نوع نوشتن در آن مکان است. یک برنامه دارای (نقص) **رقابت داده** است، هرگاه ارزیابی آن به حالتی حاوی دسترسی‌های متضاد منجر شود. به عبارتی دقیق‌تر، برنامه $P = defn^*\ e$ دارای رقابت داده است، هرگاه $P \vdash \langle \emptyset, (\texttt{main}, e)\rangle \hookrightarrow^* S$ به‌طوری که S حاوی دسترسی‌های متضاد باشد.

در ادامه نشان می‌دهیم که اگر نتیجه تحلیل اجرای مجرد برای برچسب نهایی متد اصلی یک برنامه (همانند قبل، عبارت آغازین آن برنامه در داخل متدی در یک کلاس) ایمن باشد، آن‌گاه برنامه یاد شده خالی از رقابت داده است. در همین ابتدا لازم است تأکید کنیم که فرض ما بر این است که برنامه خالی از نقص‌هایی مانند تبدیل نوع نامعتبر، شروع مجدد ریسه، و خطای دسترسی به محتوای اشاره‌گر تهی است. همچنین، از آنجایی که به دنبال اثبات درستی سیستم (یعنی فیلدهایی که با اصلاح کننده `norace` حاشیه‌نویسی شده‌اند باید دچار خطای رقابت داده نشوند) هستیم، فرض می‌کنیم که همه فیلدهای کلاس‌ها با اصلاح کننده `norace` حاشیه‌نویسی شده‌اند (توجه کنید که از نوشتن آن‌ها خودداری کرده‌ایم.) سایر فرضیات ما، در اثبات درستی یک سیستم کشف رقابت داده مبتنی بر تحلیل اجرای مجرد بین‌رویه‌ای، به شرح زیر است:

- فرض می‌کنیم که تحلیل‌های گراف فراخوانی و اشاره‌گر درست هستند. همچنین، فرض می‌کنیم که تحلیل اشاره‌گر مورد استفاده مستقل از تحلیل اجرای مجرد است. بنابراین، تابع $i_۱$ در تعریف توابع $inst$ و $inst'$ (در بخش ۳.۳.۴) را همانی در نظر می‌گیریم.

- فرض می‌کنیم که تحلیل‌های گراف فراخوانی و اشاره‌گر در تمام مراحل ارزیابی یک برنامه (به‌صورت درست) قابل استفاده هستند. به عبارتی دقیق‌تر، اگر $S = \langle \sigma, (\texttt{main}, e_۱).\ldots.(\tau_n, e_n)\rangle$ یک حالت از یک اجرای برنامه‌ای دلخواه بوده، و $[e..md(e^*)]^{\ell_c}_{\ell_r}$ یک مکان فراخوانی متد در یکی از عبارت‌های $e_۱$ تا e_n باشد، مقدار $callees(\ell_c)$ برابر است با مجموعه اسامی کامل تمام متدهایی که ممکن است در مکان مورد نظر (در حالتی که S به آن‌ها گذر می‌کند) فراخوانی شوند (همین فرض را برای مکان‌های ایجاد ریسه می‌کنیم.) همچنین، اگر e یک زیرعبارت از عبارت‌های $e_۱$ تا e_n باشد، $mayPointsTo(e)$ برابر است با مجموعه تمام شی‌های مجردی (گره‌های گراف شکل هیپ) که نشان دهنده اشیاء متمایزی هستند ممکن است (در حالتی که S به آن‌ها گذر می‌کند) نتیجه ارزیابی e برابر با یکی از آن‌ها باشد. حال اگر e' زیرعبارتی دیگر از دنباله عبارت‌های یاد شده باشد، و اگر نتیجه e و e' برابر باشند، آنگاه $mayPointsTo(e) \cap mayPointsTo(e') \neq \emptyset$. در نهایت فرض می‌کنیم که $mayPointsTo$ برای مجموعه آدرس‌ها یک به یک بوده و هر آدرس را به یک مجموعه تک نقطه‌ای نگاشت می‌کند.

- محاسبات بدون در نظر گرفتن برچسب‌ها انجام می‌شود، هنگامی که قرار است اجرای مجردی را به یک حالت نسبت دهیم، تمام عبارات ظاهر شده در حالت مورد نظر را (به‌صورت منحصر به‌فرد) برچسب‌گذاری می‌کنیم، و فرض می‌کنیم که تابع $getLabMult$ به ازای هر برچسب ظاهر شده در یک حالت مقدار یک را محاسبه می‌کند.

- فرض می‌کنیم که خلاصه متد برای تمام متدهای موجود در هر برنامه مورد بررسی موجود است (قضیه ۱.۴ برقرار بودن این شرط را تضمین می‌کند).

- برای ساخت گراف جریان کنترل (به منظور تحلیل) یک عبارت، ابتدا آن را به عنوان بدنه متدی در داخل یک کلاس در نظر می‌گیریم، به‌گونه‌ای که هیچ متد و کلاسی هم‌نام با آن‌ها در برنامه مورد بررسی موجود نباشد. با ساخت گراف جریان کنترل برای آن متد می‌توان با استفاده از تابع $IntAnalyse$ برای محاسبه جواب تحلیل برای متد مورد نظر

(و در نتیجه عبارت یاد شده) استفاده کرد. توجه کنید که از آنجایی که نام متد و کلاسی که برای تحلیل یک عبارت در نظر می‌گیریم برابر با نام هیچ متد یا کلاس دیگر در برنامه نیست، این متد فرضی با هیچ متد دیگر از برنامه مورد بررسی (و خود متد فرضی) در یک گروه نیست.

- تابع تجرید شی، α، با دریافت یک شی آن را به یک شی مجرد نگاشت می‌کند. فرض ما بر این است که نگاشت آدرس‌ها به شی‌های مجرد به‌صورت یک به یک است. این شی مجرد برای یک آدرس p در حالتی با مخزن σ طوری محاسبه می‌شود که داشته باشیم:

$$mayPointsTo(p)\{\alpha(\sigma(p))\}.$$

توجه کنید که پیش‌تر فرض کرده‌ایم که تحلیل اشاره‌گر طوری برای در نظر گرفتن عبارت‌های میانی توسعه پیدا کرده است که مقدار $mayPointsTo$ برای یک آدرس برابر با یک تک نقطه‌ای است.

- در صورتی که $S = \langle \sigma, (\texttt{main}, e_1).\ldots.(\tau_n, e_n)\rangle$ یک حالت از ارزیابی برنامه P با خلاصه متدهای ms باشد، اجرای منتسب به این حالت به‌صورت زیر تعریف می‌شود:

$$\eta = \bigsqcup\{inst(\tau_i, IntAnalyse(P, c.md, ms)(\bullet)(final(e_i)) \mid i \leq n \wedge \tau_1 = \texttt{main}\}$$

به‌طوری که $c.md$ نام متد فرضی برای عبارت e_i است. اجرای مجرد منتسب به یک حالت را با $S :: \eta$ نشان می‌دهیم، و S را ایمن گوییم هرگاه η ایمن باشد.

- فرض می‌کنیم تحلیل اجرای مجرد برای تحلیل زمینه‌های ارزیابی و عبارت‌های میانی گسترش یافته است. برای این منظور توابع لازم برای ساخت گراف جریان کنترل را به‌صورت زیر توسعه می‌دهیم (توسعه سایر توابع و تعاریف به راحتی قابل انجام است):

$$\mathcal{E} ::= [\{\}]^{\ell_h} \mid [\mathcal{E}.fn]^{\ell} \mid \cdots \mid [insync\ p\ in\ \mathcal{E}]^{\ell_n}_{\ell_x} \mid \ldots$$

$$\begin{array}{cc}
\widehat{final}([\{\}]^{\ell_h}) = \ell_h, & \widehat{init}([\{\}]^{\ell_h}) = \ell_h \\
\widehat{final}([\mathcal{E}.fn]^{\ell}) = \ell, & \widehat{init}([\mathcal{E}.fn]^{\ell}) = \widehat{init}(\mathcal{E}), \\
\vdots & \vdots \\
\widehat{final}([insync\ p\ in\ \mathcal{E}]^{\ell_n}_{\ell_x}) = \ell_x & \widehat{init}([insync\ p\ in\ \mathcal{E}]^{\ell_n}_{\ell_x}) = \ell_n \\
\vdots & \vdots
\end{array}$$

$$\widehat{flows}([\{\}]^{\ell_h}) = \emptyset,$$

$$\widehat{flows}([\mathcal{E}.fn]^{\ell}) = \{(\widehat{final}(\mathcal{E}), \ell)\} \cup \widehat{flows}(\mathcal{E}),$$

$$\widehat{flows}([\mathcal{E}.fn = e]^{\ell}) = \{(\widehat{final}(\mathcal{E}), init(e)), (final(e), \ell)\} \cup flows(e) \cup \widehat{flows}(\mathcal{E}),$$

$$\widehat{flows}([p.fn = \mathcal{E}]^{\ell}) = \{(\widehat{final}(p), \widehat{init}(\mathcal{E}), (\widehat{final}(\mathcal{E}), \ell)\} \cup \widehat{flows}(\mathcal{E}),$$

$$\vdots$$

$$\widehat{flows}([insync\ p\ in\ \mathcal{E}]^{\ell_n}_{\ell_x}) = \{(\ell_n, \widehat{init}(p)), (\widehat{final}(p), \widehat{init}(\mathcal{E})), (\widehat{final}(\mathcal{E}), \ell_x)\} \cup \widehat{flows}(\mathcal{E}),$$

$$\vdots$$

$$\widehat{blocks}([insync\ p\ in\ \mathcal{E}]^{\ell_n}_{\ell_x}) = \widehat{blocks}(\mathcal{E}) \cup \{[enterMonitor(p; mathcalE)]^{\ell_n}, [exitMonitor(p; mathcalE)]^{\ell_x}\},$$

$$\vdots$$

با استفاده از این توابع گراف جریان کنترل را برای یک زمینه ارزیابی به‌دست می‌آوریم. حال تابع انتقال برای یک سوراخ را به‌صورت زیر تعریف می‌کنیم:

$$AEA^{\mathsf{int}}_{\bullet}(\ell_h) = AEA^{\mathsf{int}}_{\circ}(\ell_h) \sqcup H \qquad ; [\{\}]^{\ell_h} \in \widehat{blocks}_{\star},$$

همان‌طور که مشاهده می‌شود، مقدار تابع انتقال برای سوراخ خود تابعی از H است. بنابراین، می‌توان نتیجه تحلیل عبارتی را که سوراخ را پر می‌کند با H جایگزین کرد. در نهایت لازم به یادآوری است که تابع انتقال برای آدرس‌ها (عبارت‌ها به فرم $[p]^{\ell_p}$) هیچ رخدادی تولید نمی‌کند، و همان‌طور که مشاهده می‌شود عبارهای $[insync\ p\ in\ \mathcal{E}]^{\ell_n}_{\ell_x}$ با استفاده از توابع انتقال موجود قابل رسیدگی هستند.

تعریف ۸.۲. اگر S یک حالت به فرم $\left\langle \sigma, T.(\tau, \mathcal{E}\left[[p.fn]^{\ell}\right]).T' \right\rangle$ باشد، آنگاه رخداد $R(\tau\ell, \tau^{\backprime}, \{\alpha(\sigma(p))\})$ مربوط به دسترسی τ است. به‌طور مشابه، اگر S یک حالت به فرم $\left\langle \sigma, T.(\tau', \mathcal{E}\left[[p.fn = v]^{\ell}\right]).T' \right\rangle$ باشد، آنگاه رخداد $W(\tau'\ell, \tau'^{\backprime}, \{\alpha(\sigma(p))\})$، رخداد مربوط به دسترسی τ' است. ■

با توجه به توابع انتقال مربوط به این دسته از عبارت‌ها در تحلیل اجرای مجرد ارائه شده در بخش ۳.۳.۴، به راحتی قابل مشاهده است که رخدادهای مربوط به این عبارات (آن‌طور که در تعریف بالا معرفی شده است) برابر با رخدادهایی است که از نمونه‌سازی رخدادهای تولید شده توسط توابع انتقال یاد شده توسط تابع $inst(\tau)$ یا $inst(\tau')$ به‌دست آمده است. هسته اصلی اثبات درستی سیستم کشف رقابت داده را دو قضیه ایمنی (۶.۲) و نگهداری (۵.۲) تشکیل می‌دهد. برای اثبات این دو قضیه نیاز به چند لم کمکی داریم که در ادامه به معرفی هر یک می‌پردازیم.

لم ۱.۲. نام متد دلخواه $c.md$ مفروض است، به‌طوری که $bodyOf(c.md) = \ldots \{e\}$ بوده، و با فرض برچسب دلخواه ℓ با چندگانگی یک و

$$(E, \mapsto) = inst'(\ell, s) \qquad ; s \in ms(c.md),$$

داریم: $\forall e' \in E \cdot \neg mwr(e')$. در این صورت با فرض شناسه دلخواه τ^m، به‌قسمی که $m \neq *$ و نیز

$$(E', \mapsto') = inst'(\tau^m, IntAnalyse(P, c.md, ms)(\bullet)(final(e)),$$

داریم:

$$\forall e' \in E' \cdot \neg mwr(e').$$

اثبات. با توجه به توابع انتقال تحلیل اجرای مجرد تمامی توابع به غیر از توابع انتقال مربوط به فراخوانی متدها و ایجاد ریسه‌ها، مؤلفه مربوط به شناسه ریسه رخدادها را برابر با $\Lambda^?$ انتخاب می‌کنند. بنابراین، وقتی که در تابع $inst'(\tau^m)$ این مؤلفه‌ها را با یک شناسه ریسه با چندگانگی به غیر از $*$ نمونه‌سازی می‌کنیم، با توجه به تعریف عملگر $\otimes$ نتیجه می‌گیریم که حکم قضیه برای تمام رخدادهای تولید شده توسط این دسته از توابع انتقال برقرار است.

حال باید ثابت کنیم که حکم قضیه برای تمام رخدادهای تولید شده توسط توابع انتقال مربوط به عبارت‌های فراخوانی متدها و ایجاد ریسه‌ها نیز برقرار است. ما صرفاً تابع انتقال مربوط به فراخوانی متدها را بررسی می‌کنیم. استدلال در مورد تابع انتقال برای عبارت‌های ایجاد ریسه به‌صورت مشابه انجام می‌شود.

از بین دو تابع انتقال برای یک مکان فراخوانی، تابعی که رخدادی تولید می‌کند، تابع انتقال مربوط به نقطه بازگشت متدها است. با مراجعه به بخش ۳.۳.۴، این تابع را به‌صورت زیر تعریف کرده‌ایم:

$$f^{ret}_{\ell_c, \ell_r}(\eta, \eta') = \eta \sqcup inst'(\ell_c, \eta' \sqcup R),$$
$$R = \bigsqcup\{s \mid ms(m) = \{s\} \wedge m \in callees(\ell_c) \wedge (m, c'.md') \notin SCC\}.$$

که در آن $c'.md'$ نام متد فرضی انتخاب شده برای عبارت e است. طبق فرض خود برای ساخت گراف جریان کنترل برای عبارت‌ها، نام متد $c'.md'$ نام هیچ متد در برنامه P نیست. بنابراین، با هیچ متد موجود در این برنامه رابطه SCC ندارد. در نتیجه طبق تعریف گراف جریان کنترل بین‌رویه‌ای، هیچ کمان بین‌رویه‌ای وارد گره ℓ_r نمی‌شود. بنابراین، η' برابر با $\bot$ بوده، و از آنجایی که فرض کرده‌ایم خلاصه تمام متدهای برنامه موجود است، تمام رخدادهای تولید شده توسط تابع یاد شده از نمونه‌سازی رخدادهای موجود در مجموعه R با برچسب ℓ_c با چندگانگی یک به‌دست می‌آید. در نتیجه کافی است ثابت کنیم که نمونه‌سازی خلاصه هر یک از متدهای موجود در $mayCall(c'.md')$ هیچ رخداد نوشتن چندگانه‌ای تولید نمی‌کند. برای این منظور از آنجایی که $mayCall(c.md) = mayCall(c'.md')$ است، کافی است ثابت کنیم که اگر m' عنصری دلخواه از $mayCall(c.md)$ باشد، و نیز ℓ' برچسبی دلخواه با چندگانگی یک باشد، با فرض $(E'', \mapsto'') = inst'(\ell', s)$ به ازای $s \in ms(m')$ داریم:

$$\forall e'' \in E'' \cdot \neg mwr(e'').$$

با توجه به این که برای هر m' در $mayCall(m')$ یک برچسب ℓ'_c در مجموعه برچسب‌های بدنه متد $c.md$ موجود است، به قسمی که:

$$inst'(\ell'_c, s) \sqsubseteq (E, \mapsto) \qquad ; s \in ms(m')$$

حال، طبق تعریف رابطه $\sqsubseteq$ می‌توان نتیجه گرفت $E'' \subseteq E$ و از آنجا با توجه به فرض قضیه نتیجه می‌گیریم:

$$\forall e' \in E'' \cdot \neg mwr(e').$$

■

لم ۲.۲. نام متد دلخواه $c.md$ مفروض است، به‌طوری که $bodyOf(c.md) = \ldots\{e\}$ و با فرض برچسب دلخواه ℓ با چندگانگی یک و

$$(E, \mapsto) = inst'(\ell, s) \qquad ; s \in ms(c.md),$$

داریم:

$$\forall e'_1, e'_2 \in E \cdot conf(e'_1, e'_2) \implies ord_{(E,\mapsto)}(e'_1, e'_2).$$

در این‌صورت برای یک شناسه دلخواه τ^m با فرض

$$(E', \mapsto') = inst'(\tau^m, IntAnalyse(P, c.md, ms)(\bullet)(final(e)),$$

داریم:

$$\forall e'_1, e'_2 \in E' \cdot conf(e'_1, e'_2) \implies ord_{(E',\mapsto')}(e'_1, e'_2).$$

اثبات. تمام توابع انتقال در تحلیل اجراهای مجرد ارائه شده در بخش ۳.۳.۴ به غیر از تابع انتقال مربوط به ایجاد ریسه‌ها، رخدادهایی با مؤلفه شناسه ریسه برابر با $\Lambda^?$ تولید می‌کنند. بنابراین، اگر این رخدادها را با یک شناسه ریسه مانند τ^m نمونه‌سازی کنیم، طبق تعریف رابطه ord به راحتی می‌توان مشاهده کرد که در این رابطه شرکت می‌کنند.

همانند اثبات لم ۱.۲ باید ثابت کنیم که حکم قضیه برای تمام رخدادهای به‌دست آمده از خلاصه متدهای $c.\mathrm{run}$ که در مکان‌های ایجاد ریسه استفاده می‌شوند برقرار است. همانند قبل، از آنجایی که برای هر m' در $mayCall(c.md)$ یک برچسب ℓ_f یا ℓ_c در بدنه متد $c.md$ موجود است، به قسمی که

$$inst(\ell_f, s) \sqsubseteq (E, \mapsto) \qquad ; s \in ms(m'),$$

یا

$$inst'(\ell_c, s) \sqsubseteq (E, \mapsto) \qquad ; s \in ms(m').$$

حال می‌توان با فرض $(E'', \mapsto'') = inst(\ell_f, s)$ نتیجه گرفت:

$$\forall e''_۱, e''_۲ \in E'' \cdot conf(e''_۱, e''_۲) \implies ord_{(E'', \mapsto'')}(e''_۱, e''_۲).$$

در نهایت با توجه با اینکه رخدادهای تولید شده توسط تحلیل برای عبارت e با رخدادهای موجود در خلاصه متد $c.md$ تنها در شناسه رخدادها تفاوت دارد، نتیجه می‌گیریم که با توجه به فرض قضیه، حکم برای رخدادهای تولید شده برای عبارت e و نیز رخدادهای حاصل از ایجاد ریسه‌ها در عبارت یاد شده برقرار است. ■

تعریف ۹.۲. در صورتی که $S = \langle \sigma, T.(\tau, e).T' \rangle$ یک حالت باشد، که در آن e به یکی از فرم‌های $\mathcal{E}\left[[p.fn]^\ell\right]$ یا $\mathcal{E}\left[[p.fn = v]^\ell\right]$ باشد، مجموعه زیر را

$$L^p_\tau = \{p' \mid \ldots insync\ p'\ in\ \ldots [e']^\ell \cdots \in e\},$$

که در آن e' بسته به نوع دسترسی e می‌تواند به فرم $p.fn$ یا $p.fn = v$ باشد، مجموعه تمام قفل‌هایی که حین دسترسی به آدرس p توسط τ اخذ می‌شود گوییم. ■

لم ۳.۲. اگر S یک حالت دلخواه باشد که در آن (عبارت‌های مربوط به) دو ریسه τ و τ'، به‌طوری که $\tau \neq \tau'$، به آدرس p دسترسی داشته باشند، و $\eta :: S$ باشد، آنگاه رخدادهای مربوط به این دو دسترسی با هم رابطه ord_η ندارند.

اثبات. از برهان خلف برای اثبات این قضیه استفاده می‌کنیم. فرض می‌کنیم که دو رخداد یاد شده با هم رابطه ord_η دارند. در این‌صورت طبق تعریف این رابطه دو حالت را باید بررسی کرد:

۱- دو رخداد مربوط به یک ریسه هستند (یعنی شناسه رخدادهای آن‌ها با هم برابر است)،

۲- دو رخداد رخداد مربوط به ریسه‌های مختلف هستند، و از طریق عملیات ورود و خروج از مانیتور مرتب شده‌اند.

حالت اول بلافاصله با توجه به فرض قضیه، که تصریح می‌کند که عملیات دسترسی تصریح توسط ریسه‌های مختلف صورت گرفته است، رد می‌شود. حالت دوم طبق تعریف رابطه ord_η به دو بخش تقسیم می‌شود. اول این است دو رخداد خود عملیات مانیتور باشند که بر هم منطبق می‌شوند (جفت هم هستند): این حالت با توجه به فرض قضیه که بیان می‌کند که عملیات مورد نظر از نوع دسترسی هستند رد می‌شود. حالت دوم این است که دو رخداد دسترسی یاد شده توسط رخدادهای مانیتوری که بر هم منطبق هستند محاصره شده است. این یعنی دو ریسه، ضمن دسترسی به آدرس p، حداقل یک قفل مشترک را اخذ کرده‌اند. به عبارتی دقیق‌تر برای حالت S داریم:

$$L^p_\tau \cap L^p_{\tau'} \neq \emptyset.$$

با توجه به دوگان لم ۴.۲، از این نکته که مجموعه قفل‌های اخذ شده توسط جفت ریسه τ و τ' اشتراکی با هم ندارند، نتیجه می‌گیریم که در حالت S دو دسترسی توسط جفت ریسه یاد شده صورت نگرفته است، که با فرض قضیه مبنی بر دسترسی دو ریسه مورد نظر به مکان p در تناقض است. در نتیجه فرض خلف باطل و می‌توان گفت رخدادهای مربوط به دو دسترسی با هم رابطه ord_η ندارند. ■

لم ۴.۲. حالت S و دو ریسه دلخواه τ و τ'، به‌طوری که $\tau \neq \tau'$، مفروض است. در صورتی که (عبارت‌های مربوط به) دو ریسه یاد شده به ترتیب به آدرس‌های p_1 و p_2 دسترسی داشته باشند، آن‌گاه $L_\tau^{p_1} \cap L_{\tau'}^{p_2} = \emptyset$.

اثبات. برای اثبات این قضیه فرض می‌کنیم اشتراک این دو مجموعه غیر تهی باشد، در این صورت یک حالت S' یا S'' می‌توان یافت، به قسمی که

$$S' = \langle \sigma, T_1.(\tau, \mathcal{E}\,[\texttt{synchronized}\ p\ \texttt{in}\ e]).T_2 \rangle \hookrightarrow^* S_i \hookrightarrow^* S,$$

یا

$$S'' = \langle \sigma', T'_1.(\tau', \mathcal{E}\,[\texttt{synchronized}\ p\ \texttt{in}\ e_2]).T'_2 \rangle \hookrightarrow^* S_i \hookrightarrow^* S$$

و $p \in L_\tau^{p_1} \cap L_{\tau'}^{p_2}$ است. همچنین، هیچ حالت S_i به فرم‌های

$$S_i = \langle \sigma'', T''_1.(\tau, insync\ p\ in\ v).T''_2 \rangle,$$

یا

$$S_i = \langle \sigma''', T'''_1.(\tau', insync\ p\ in\ v).T'''_2 \rangle,$$

موجود نیست. ما به بررسی S' بسنده می‌کنیم، و بررسی S'' به‌صورت مشابه انجام می‌شود. با توجه به قاعده [Red Sync] و [Red InSync] در صورتی که در حالت S' داشته باشیم $\sigma(p) = (\!|db|\!)_c^{\mathsf{unlocked}}$ حالت S' به حالتی گذار می‌کند که در آن وضعیت قفل p بسته است، و بعد از آن با توجه به پیش‌شرط قاعده [Red Sync] هیچ عبارت به فرم $\texttt{synchronized}\ p\ \texttt{in}\ e_2$ از هر ریسه دلخواه قابل ارزیابی نیست. در نتیجه داریم:

$$p \notin L_{\tau'}^{p_2} \tag{۱.۲}$$

بر عکس زمانی که $\sigma(p) = (\!|db|\!)_c^{\mathsf{locked}}$ است، با توجه به پیش‌شرط قاعده [Red Sync] عبارت مربوط به ریسه τ از ارزیابی بیشتر بازمانده و این‌بار داریم:

$$p \notin L_\tau^{p_1} \tag{۲.۲}$$

با توجه به (۱.۲) و (۲.۲) نتیجه می‌گیریم که فرض خلف باطل و داریم $L_\tau^{p_1} \cap L_{\tau'}^{p_2} = \emptyset$. ■

حال به اثبات دو قضیه مهم که به‌طور مستقیم برای اثبات درستی سیستم به‌کار می‌روند را مورد بررسی قرار می‌دهیم. ابتدا قضیه نگهداری و سپس قضیه پیش‌روی را ارائه می‌دهیم.

قضیه ۵.۲ (نگهداری). اگر S یک حالت ایمن باشد، و $S \hookrightarrow S'$، آن‌گاه S' نیز ایمن است.

اثبات. اثبات این قضیه را با استقرا بر روی ساختار $S \hookrightarrow S'$ که در شکل ۱.۲ توصیف شده است، بیان می‌کنیم:

قاعده [**Red New**]: با توجه فرض استقرا و نیز با توجه به این‌که تابع انتقال مربوط به شناسه شی‌ها هیچ رخدادی تولید نمی‌کند، حکم برقرار است.

قاعده [**Red Read**]: با توجه فرض استقرا و نیز با توجه به این‌که تابع انتقال مربوط به مقادیر هیچ رخدادی تولید نمی‌کند، حکم برقرار است.

قاعده [**Red Assign**]: با توجه فرض استقرا و نیز با توجه به این‌که تابع انتقال مربوط به مقادیر هیچ رخدادی تولید نمی‌کند، حکم برقرار است.

قاعده [**Red Call**]: فرض می‌کنیم که $S = \langle \sigma, T.(\tau, e_i).T' \rangle$، به‌طوری که

$$e_i = \mathcal{E}\left[[p.mn(v_1, \ldots, v_n)]_{\ell_r}^{\ell_c}\right],$$

ایمن است. می‌خواهیم ثابت کنیم، که با فرض $bodyOf(c.mn) = t\ mn(x_1, \ldots, x_m)\{e\}$ و $\sigma(p) = (\!|db|\!)_c^m$، حالت

$$S' = \langle \sigma, T.(\tau, e_j).T' \rangle$$

با فرض

$$e_j = \mathcal{E}\left[e\left[v_k/{x_k}^{k\in\{1,\ldots,m\}}, p/\texttt{this}\right]\right]$$

نیز ایمن است.

همان‌طور که مشاهده می‌شود، تنها تفاوت دو حالت S و S' در دنباله ریسه‌ها و در عبارتی است که در سوراخ موجود در زمینه ارزیابی مربوط به ریسه τ جای‌گذاری می‌شود. طبق فرض استقرا S ایمن است. بنابراین، اجرای مجرد:

$$inst'(\tau, IntAnalysis(c'.mn')(\bullet)(\ell_r)), \tag{۳.۲}$$

که در آن $c'.mn'$ نام متد فرضی است که در برگیرنده عبارت $[p.mn(v_1, \ldots, v_n)]_{\ell_r}^{\ell_c}$ است، ایمن است. حال با فرض e به عنون بدنه متد فراخوانی شده، طبق لم‌های ۱.۲ و ۲.۲ داریم:

$$inst'(\tau, IntAnalysis(c''.mn'')(\bullet)(final(e))), \tag{۴.۲}$$

که در آن $c''.mn''$ نام متد فرضی است که در برگیرنده عبارت $e\left[v_k/{x_k}^{k\in\{1,\ldots,m\}}, p/\texttt{this}\right]$ باشد نیز ایمن است. از آنجایی که تفاوت رخدادهای موجود در ۳.۲ و ۴.۲ تنها در شناسه رخدادها است، نتیجه می‌گیریم که اجرای مجرد منتسب به عبارت e_j و در نتیجه اجرای منتسب به حالت S' ایمن است.

قاعده [**Red Let**]: با توجه فرض استقرا و نیز با توجه به این‌که تابع انتقال مربوط به ورود و خروج از بلوک `let` هیچ رخدادی تولید نمی‌کند، حکم برقرار است.

قاعده [**Red Sync**]: با توجه فرض استقرا و نیز با توجه به این‌که توابع انتقال مربوط به ساختارهای `synchronized` و $insync$ با هم برابر هستند، حکم برقرار است.

قاعده [**Red Start**]: فرض می‌کنیم $S = \langle \sigma, T.(\tau, e_s).T' \rangle$، به‌طوری که $e_s = [p.\texttt{start}()]^{\ell_f}$، ایمن است. می‌خواهیم ثابت کنیم که حالت $S' = \langle \sigma, T.(\tau, e_n).T'.(\tau', e_r) \rangle$، با فرض:

$$e_n = \mathcal{E}\,[\texttt{null}]\,,$$

$$e_r = p.\texttt{run}(),$$

$$S \hookrightarrow S',$$

ایمن است.

از آنجایی که عبارت `null` رخدادی تولید نمی‌کند، با توجه به فرض استقرا داریم حالت:

$$S'' = \langle \sigma, T.(\tau, \mathcal{E}\,[\texttt{null}]).T' \rangle\,,$$

نیز ایمن است. همچنین، با توجه به فرض استقرا اجرای مجرد:

$$inst(\tau, IntAnalyse(c'.mn')(\bullet)(\ell_f)), \tag{۵.۲}$$

که در آن $c'.mn'$ نام متد فرضی در برگیرنده عبارت $[p.\texttt{start}()]^{\ell_f}$ است، ایمن است. از آنجایی که رخدادهای تولید شده در توسط عبارت $p.\texttt{run}()$ در ریسه τ' با رخدادهای تولید شده در ۵.۲ تنها در شناسه رخدادها با هم تفاوت دارند، نتیجه می‌گیریم که

$$inst(\tau', IntAnalyse(c''.mn'')(\bullet)(\ell_r))$$

به طوری که $c''.mn''$ نام متد فرضی در برگیرنده عبارت $[p.\texttt{run}()]^{\ell_c}_{\ell_r}$ است، نیز ایمن است. در نتیجه حالت S' ایمن است.

قاعده [**Red Cast**]: با توجه به فرض استقرا بلافاصله نتیجه می‌گیریم که حکم برقرار است. ■

قضیه ۶.۲ (ایمنی). اگر S یک حالت باشد، به‌طوری که $S :: \eta$ و η ایمن باشد، آنگاه S حاوی دسترسی‌های متضاد نیست.

اثبات. فرض می‌کنیم که S دارای رقابت داده باشد، یعنی حالت مورد نظر، با فرض $\tau \neq \tau'$، به یکی از فرم‌های زیر باشد:

۱- $S = \langle \sigma, T.(\tau, \mathcal{E}\,[p.fn]).T'.(\tau', \mathcal{E}\,[p.fn = v]).T'' \rangle$ یا،

۲- $S = \langle \sigma, T.(\tau, \mathcal{E}\,[p.fn = v_۱]\,.T'.(\tau', \mathcal{E}\,[p.fn = v_۲]).T'' \rangle$.

از آنجایی که زیرعبارت‌های $p.fn$ و $p.fn = v$ در گراف جریان کنترل مربوط به عبارت‌های $\mathcal{E}[p.fn]$ و $\mathcal{E}[p.fn = v]$ به‌صورت دو گره در نظر گرفته می‌شود، و تحلیل اجرای مجرد دو رخداد برای این عبارت‌ها استخراج می‌کند. با توجه به توابع انتقال ارائه شده در بخش ۳.۳.۴ برای عبارت‌های دسترسی به محتوا و انتساب واضح است که رخدادهای تولید شده توسط آن‌ها بعد از نمونه‌سازی توسط تابع $inst(\tau)$ و $inst(\tau')$ برابر با رخدادهای مربوط به دسترسی‌های یاد شده است. در نتیجه رخدادهای مربوط به دو دسترسی متضاد که با هم رابطه $conf$ دارند، و در مجموعه رخدادهای اجرای مجرد η (مؤلفه اول آن) موجود هستند. از طرفی طبق لم ۳.۲ دو رخداد مربوط به دسترسی‌های موجود در حالت S با هم رابطه ord_η ندارند. بنابراین، η ایمن نیست و فرض خلف باطل می‌شود. ■

حال تعریفی از یک اجرا را معرفی می‌کنیم تا با کمک آن نتیجه قضایای بیان شده تا کنون را ارائه دهیم.

تعریف ۱۰.۲. یک اجرا از برنامه $P = defn^*\ e$ عبارت است از:

۱- یک دنباله متناهی از حالات $S_1, S_2, \ldots, S_k$ به‌طوری که $S_1 = \langle \emptyset, (\texttt{main}, e)\rangle$ و $S_n = \langle \sigma, (\texttt{main}, v_1), \ldots, (\tau_n, v_n)\rangle$ و به ازای هر $0 < i < k$ داریم $S_i \hookrightarrow S_{i+1}$؛ یا،

۲- یک دنباله نامتناهی از حالات $S_1, S_2, \ldots$ به‌طوری که $S_1 = \langle \emptyset, (\texttt{main}, e)\rangle$ برای هر $i > 0$ داریم $S_i \hookrightarrow S_{i+1}$.

■

نتیجه ۱.۲. برنامه $P = defn^*\ e$ مفروض است. در صورتی که η اجرای مجرد منتسب به گره نهایی e (عبارت آغازین برنامه P) باشد، و η ایمن باشد، آنگاه در هر اجرا از این برنامه تمامی حالت‌های آن اجرا ایمن هستند. به عبارتی دقیق‌تر اگر η ایمن باشد، و دنباله $\langle S_i\rangle$ اجرایی دلخواه از P باشد به ازای هر i داریم S_i ایمن است.

اثبات. اثبات این قضیه به دو بخش تقسیم می‌شود: در بخش اول حکم قضیه را برای اجراهای متناهی اثبات می‌کنیم، و در بخش دوم این حکم را برای اجراهای نامتناهی بررسی می‌کنیم. طبق تعریف ۱۰.۲ مشاهده می‌کنیم که اولین عنصر هر اجرا حالت $S_1 = \langle \emptyset, (\texttt{main}, e)\rangle$ است، و با توجه به این که η (اجرای مجرد منتسب به گره نهایی عبارت آغازین برنامه مورد بررسی) ایمن است، اجرای مجرد منتسب به حالت S_1 نیز ایمن است. حال برای اجراهای متناهی بر روی اندازه اجرا استقرا می‌زنیم:

اجرا به‌صورت $\langle S_1\rangle$ است: بدیهی است که هر حالت موجود در این دنباله ایمن است.

فرض استقرا: فرض می‌کنیم که هر حالت موجود در اجرایی مانند $\langle S_1, \ldots, S_k\rangle$ برای $k > 1$ ایمن است.

اجرا دنباله‌ای به‌صورت $\langle S_1, \ldots, S_k, S_{k+1}\rangle$ است: با توجه به فرض استقرا هر حالت در $\langle S_1, \ldots, S_k\rangle$ ایمن است. از طرفی با توجه به تعریف ۱۰.۲ داریم: $S_k \hookrightarrow S_{k+1}$. با توجه به قضیه ۵.۲ نتیجه می‌گیریم که S_{k+1} نیز ایمن است.

برای اثبات بخش دوم قضیه از برهان خلف استفاده می‌کنیم: اگر $\langle S_1, S_2, \ldots\rangle$ یک اجرا (نامتناهی) از برنامه مورد نظر باشد، فرض می‌کنیم عددی مانند $k > 1$ موجود است به قسمی که S_k از دنباله یاد شده ایمن نیست. با توجه به اینکه S_1 ایمن است، و داریم $S_1 \hookrightarrow^* S_k$، با k بار اعمال قضیه ۵.۲ نتیجه می‌گیریم که فرض خلف باطل است. ■

واژه‌نامه

رقابت داده	Data-race
دنباله آثار	Trace
الگوی اشکال	Bug Pattern
همگام‌سازی	Synchronization
پیش‌رویدادی	Happens-before
تحلیل جریان داده	Data-flow Analysis
زمینه	Context (-sensitive)
دگرنام (تحلیل دگرنامی)	Alias (Analysis)
سیاست قفل‌گذاری	Locking Discipline
دستورالعمل	Statement/Instruction
همروندی (تحلیل همروندی)	Concurrency (Analysis)
رقابت شی	Object-race
نوع (وارسی نوع)	Type (Checking)
برنامه مشتری	Client Program
وب‌گاه	Website
پیوند	Join
قید (تحلیل مبتنی بر قید)	Constraint (-based Analysis)
تحلیل ممکن‌است‌همزمان‌رخ‌دهد	May-happen-in-parallel Analysis
جایگزین کردن	Override
صافی	Filter

ریسه	Thread
اکتشافی	Hueristic
نقص	Defect
بسیط	Atomic
درست (نادرست)	Sound (Unsound)
جریان (حساس به جریان)	Flow (-sensitive)
تحلیل اشاره‌گر	Pointer Analysis
مجموعه قفل‌ها	Lockset
رخداد	Event
پیمانه‌ای	Modular
ریزدانگی	Granularity
نوع ایمن	Type-safe
مالکیت	Ownership
اصطلاح	Idiom
مشبک(مشبک تام)	Lattice (Complete Lattice)
اشتراک	Meet
تفسیر مجرد	Abstract Interpretation
تک عنصری/تک نقطه‌ای	Singleton
پشته فراخوانی	Call-stack
گردایه	Collection

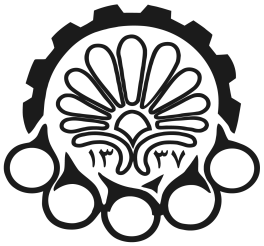
Abstract

By the advent of multi-core processors in computing devices, software developers have become more interested in concurrent programming. Concurrency, however, introduces new kinds of errors into software systems. Harmful data races are example of such errors which comes along as an aftermath of incorrect synchronization of concurrent threads on accessing shared resources. Due to the nondeterministic nature of data-race errors, tracing the exact place and the moment in which the error has been occurred is notoriously difficult. Also, in most cases, the consequences of data races could be disastrous. Hence, automated data-race detection and prevention are of particular importance.

Despite considerable advances in automated data-race detection in the last decade, a scalable tool, yet sound and precise, is still a need. In this thesis, we present a language-based technique for data-race detection. The underlying idea is realized as a data-flow analysis which is demonstrated in a core subset of Java programming language. By using the technique, one can implement tools for scalable, sound, and precise static data-race detection. Parametric events and the notion of method summary are at the heart of our idea in formulating a modular analysis method resulting in scalability. In fact, unlike many earlier methods for data-race detection, we do not build a system of equations for the whole program. Instead, each method is analyzed in isolation with a method summary as a result. In this way, we do not need to run different analyses when the same method is called at different call sites. The proposed technique can also be used for the analysis of open programs such as libraries where there is no prior knowledge of their client programs. We annotate some program constructs with type qualifiers and other information useful for the analysis. This annotation, however, is restricted to interfaces making its burden more bearable. We implement a tool to check Java programs for race freedom. Experiments show that the tool not only scales to more than ten kilo LOC but also produces results comparable to leading precise data-race detectors.

Keywords:

Data-race, Data-flow Analysis, Event, Java, Control Flow Graph, Pointer Analysis, Control Flow Analysis



Amirkabir University of Technology
(Tehran Polytechnic)

Department of Computer Engineering and Information Technology

MSc Thesis

Title

A Language-Based Approach to Prevent Data-Races

By
Ali Ghanbari

Supervisor
Mehran S. Fallah

October 2014